Cape Peninsula
University of Technology

Digital Control of a Class-D Audio Amplifier

by

Jason Quibell

Thesis submitted in fulfilment of the requirements for the degree
Master of Technology: Electrical Engineering in the Faculty of Engineering
at the Cape Peninsula University of Technology

Supervisor: Prof. R.H. Wilkinson
Co-supervisor: Prof. H. du T. Mouton

Cape Town, South Africa
November 2011

Declaration

I, Jason Quibell, declare that the contents of this thesis represent my own unaided work, and that the thesis has not previously been submitted for academic examination towards any qualification. Furthermore, it represents my own opinions and not necessarily those of the Cape Peninsula University of Technology.

Signature: ....

Date: 23 November 2011 ....

Abstract

Modern technologies have led to extensive digital music reproduction and distribution. It is fitting then that digital audio be amplified directly from its source rather than being converted to an analogue waveform before amplification. The benefits of using a digital controller for audio processing include being able to easily reconfigure the system and to add additional functions at a later stage.

Digital audio is primarily stored as Pulse Code Modulation (PCM) while Pulse Width Modulation (PWM) is the most popular scheme used to drive a class-D amplifier. The class-D amplifier is selected in many applications due to its very high energy efficiency. Conventional PCM to PWM conversion is inherently nonlinear. Various interpolation schemes are presented in this research project which help to address the nonlinearity.

Digitally generated PWM has a limited resolution which is constrained by the system clock. This thesis presents noise shaping techniques which increase the effective resolution of the PWM process without having to use an excessively high system clock. Noise shaping allows a low resolution modulator to be used to reproduce high resolution audio.

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations:

National Research Foundation (NRF)

Cape Peninsula University of Technology (CPUT)

Centre for Instrumentation Research (CIR)

French South African Institute of Technology (F'SATI)

Professor Richardt Wilkinson for his guidance

Professor Toit Mouton for his generosity of knowledge

Mr Robert Neilson for his insightful suggestions

Professor Gerhart de Jager for his years of wisdom

Charl Jooste for his camaraderie

The financial assistance of the National Research Foundation and the Cape Peninsula University of Technology towards this research is acknowledged. Opinions expressed in this thesis and the conclusions arrived at, are those of the author, and are not necessarily to be attributed to the respective mentioned establishments.

Dedications

To Mom, Dad, Bonnie, Tyrus, Keegan,
Oliver and Gaby for their unwavering support

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

ADC	Analogue-to-Digital Converter
AES/EBU	Audio Engineering Society/European Broadcasting Union
CD	Compact Disk
CIC	Cascaded Integrator Comb
CIFB	Cascade of Integrators with distributed Feedback
CIFF	Cascade of Integrators with Feedforward summation
CRFB	Cascade of Resonators with distributed Feedback
CRFF	Cascade of Resonators with Feedforward summation
DAC	Digital-to-Analogue Converter
DAT	Digital Audio Tape
DC	Direct Current (0 Hz)
DSP	Digital Signal Processing
DVD	Digital Versatile Disc
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
IIR	Infinite Impulse Response
IM	Intermodulation
LSB	Least Significant Bit
LC	Inductor-Capacitor
MAE	Minimum Aliasing Error
MIF	Memory Initialisation File
MSB	Most Significant Bit
NPWM	Natural Pulse Width Modulation
NTF	Noise Transfer Function
OSR	Over Sampling Ratio
PCM	Pulse Code Modulation
PLL	Phase Lock Loop
PMA	Pulse Modulation Amplifier
PWM	Pulse Width Modulation
ROM	Read Only Memory

RMS	Root Mean Square
SNR	Signal to Noise Ratio
S/PDIF	Sony/Philips Digital Interconnect Format
SPI	Serial Peripheral Interface
SQNR	Signal-to-Quantisation Noise Ratio
THD+N	Total Harmonic Distortion plus Noise
UPWM	Uniform Pulse Width Modulation
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

1.1 Background

The prevalence of digital audio in our modern world suggests the need for an efficient and exceptional way in which it can be amplified. All audio amplifiers are in some way or another analogue devices, as the final result will always be that of an analogue waveform (Groenenberg *et al.*, 2006). Amplifying audio from a digital source therefore requires digital-to-analogue conversion.

Most amplifier classes are very inefficient. The class-D amplifier however has a theoretical operating efficiency of 100% (Self, 2006). This class has the added advantage of switching between a fully on and fully off state, providing a straightforward means to drive it from digital circuitry. Being able to utilise these benefits offered by a class-D configuration when handling digital audio is therefore an important task.

1.2 Problem statement

The efficiency of a typical analogue amplifier is characteristically less than 40% (Self, 2006). The class-D amplifier is therefore a good substitute for a traditional amplifier when digital audio needs to be amplified due to its switching characteristic and good power efficiency. The class-D amplifier is not a digital system, yet can be better thought of as a Pulse Modulation Amplifier (PMA), where a stream of pulses are increased in amplitude (Nielsen, 1998). The audio signal is modulated onto a carrier at a low power level which is then amplified by the power stage of the class-D amplifier. Its amplified pulse is subsequently reconstructed by a low-pass filter which then results in an analogue audio waveform.

Pulse Width Modulation (PWM) is a very popular method for modulating the audio signal (Koeslag, 2008). The digital audio would typically be in a Pulse Code Modulation (PCM) format and would need to be converted to PWM. Analogue implementations of PWM are just as successful as digital ones, although using a digital controller for the signal processing allows many additional features to be added to the system's firmware, without adjusting the hardware configuration.

The process of PCM to PWM conversion is inherently nonlinear as well as resulting in low resolution audio due to system clock requirements of the PWM process (Goldberg *&* Sandler, 1991*b*).

Compact Disk (CD) audio is encoded with 16-bits at 44.1 kHz while Digital Versatile Disk (DVD) audio is capable of audio encoded with 24-bits at a sampling rate of 192 kHz. These figures quintessentially define what is described as high resolution audio. The problem is how to obtain high resolution PWM which could reproduce such audio data. Very large system clock speeds are required to generate a PWM signal which has 2^{16} distinct pulse widths per a pulse period. Needless to say, 2^{24} distinct pulses are even more difficult to achieve. Shown in Table 1.1 is a list of system clock rates needed to achieve a desired bit resolution of PWM with a switching rate of 384 kHz. The reason that 384 kHz was chosen as the switching frequency is due to it being a multiple of 48 kHz which is a sample rate used by many professional audio systems, such as Digital Audio Tape (DAT). The value of the switching frequency is selected as being multiple integer times higher than the audio sampling rate so that the switching fundamental and its harmonics would not intrude on the audio band.

Table 1.1: Required system clock rates for specific PWM resolutions switching at 384 kHz

PWM bit resolution	System clock (GHz)
7	0.049152
8	0.098304
10	0.393216
12	1.572864
16	25.165824
18	100.663296
20	402.653184
24	6442.450944

1.3 Objectives of the research

The primary objective of this research is to convert PCM to PWM while achieving outstanding audio performance. The various mechanisms which cause the audio signal to deteriorate during the conversion process need to be addressed.

The following research questions became significant:

- Can methods be introduced into the PCM to PWM conversion process which will allow for a high resolution audio output?
- Will these methods effectively linearise the conversion and allow for a realisable system clock?
- Can these methods be implemented in a real-time system on embedded hardware?

Increasing the sampling rate has been shown to improve the linearity (Goldberg & Sandler, 1991*a*). Sample rate conversion will still not make it possible to achieve high resolution audio, due to the excessively high clock rates required by the pulse width modulator. Clock rates above 200 MHz are considered to be excessively high for this project.

In order to address the problem of a very fast system clock speed, a technique called noise shaping can be employed. It is a process by which the noise floor in the audio band is reduced while still using a low resolution modulator. This would enable a slower clock speed to be used while still obtaining a high resolution audio output.

This thesis will discuss:

- Increasing the linearity of the PCM to PWM conversion process through the use of sample rate conversion,
- reducing the system clock requirements needed to duplicate a high resolution audio signal through the use of noise shaping,
- minimising the system resources required for implementation,
- simulating the various methods presented,
- developing these methods into a control algorithm with VHDL based firmware and
- evaluating the developed firmware on a tangible system.

This thesis will not discuss:

- Design or construction of a class-D power stage and
- analysis of the imperfections introduced by a class-D power stage.

1.4 Structure of the thesis

This thesis is divided into three parts. Firstly an overview of digital signal processing and pulse width modulation is presented. The primary focus of this research is then exhibited with chapters on interpolation and noise shaping. The final part of the thesis is the implementation of the firmware and the results obtained.

Chapter 2 introduces digital signal processing and its use in the modern world. Basic signal theory is highlighted to show the importance of sampling in a digital system and the effect it has on performance. The topic of digital signal processing hardware is broached and popular manufacturers are mentioned.

Chapter 3 concerns pulse width modulation and explains its operation. The difference between generating a pulse width modulated signal in the digital and analogue domains is explained. The harmonic distortion of both methods is presented.

Chapter 4 describes the process of interpolation which is used for sample rate conversion. This forms a significant part of the research. Interpolation helps to linearise the PCM to PWM process. It also increases the bandwidth, which is important for when noise shaping is utilised. Various methods are presented with their advantages and disadvantages discussed.

Chapter 5 introduces noise shaping which is the key to obtaining a high resolution audio output while using a low resolution modulator. Two distinct methods are introduced with an emphasis placed on the second, which is the use of a digital loop filter. Through the use of simulations, this technique is shown to significantly reduce the noise floor in the audio band.

Chapter 6 takes the methodology presented in the previous chapters and describes the formulation of the firmware for an embedded system. The various sections are divided into subsystem blocks.

Chapter 7 discusses problems which were discovered when moving from a simulated system to a physical implementation, and how it led to a redesign of certain subsystems.

Chapter 8 presents the results of the project. Comparisons between Matlab® and VHDL simulations are made. The test set up of the embedded real-time system is shown and its results are discussed.

Chapter 9 concludes the thesis by giving an outline of what was discussed and achieved. The possibilities for future research are also mentioned.

Chapter 2

Digital signal processing

This chapter introduces elementary digital signal processing (DSP) theory and presents an overview of its applications. Examples are used to illustrate its usefulness in modern electronic engineering.

2.1 DSP applications

The application of DSP systems in contemporary life has become invaluable. Without the ability to manipulate and transfer data in a precise and functional manner, many modern marvels would cease to exist. Tan (2008:pg. 11) mentioned just a few of the many applications which are possible with DSP systems, they include:

- Voice recognition,
- speech synthesis systems,
- image and video editing systems,
- digital electrocardiography analysers,
- digital cameras,
- digital recording,
- cellular telephones and
- wireless networks.

DSP provides engineers and scientists with many powerful tools which can be used to visualise and analyse their designs. Applications of DSP are increasing in many areas where analogue electronics are being replaced by a more effective digital counterpart.

2.2 Signal sampling and quantisation

It is not possible to digitise all points along a continuous time signal as it contains an infinite number of points. Digital processing hardware could not possibly store an infinite amount of data. Sampling solves this problem by taking samples at discrete time intervals.

The process of transforming an analogue signal into a digital sequence is known as analogue-to-digital conversion. While digital-to-analogue conversion is the reverse process. The end result of this research project is to essentially develop a high powered Digital-to-Analogue Converter (DAC).

2.2.1 Nyquist-Shannon sampling theorem

For a DSP system using uniform sampling , an analogue signal can be perfectly recovered as long as the sampling rate is at least greater than twice the bandwidth of the analogue signal to be sampled (Shannon, 1949). The Nyquist frequency is referred to as f_{max}, and the minimum sampling frequency f_s, is called the Nyquist rate.

For a given sampling interval T, which is the time between two sample points, the sampling rate is given by

$$f_s = \frac{1}{T} \text{ samples per second (Hz).} \tag{2.2.1}$$

In order to satisfy the criteria of the sampling theorem

$$f_s - f_{max} > f_{max}. \tag{2.2.2}$$

Solving equation (2.2.2) gives

$$f_s > 2f_{max}. \tag{2.2.3}$$

As an example, sampling an audio signal which possesses frequencies up to 20 kHz, f_s would need to be at least 40 kHz (Tan, 2008). If the Nyquist criteria have been adhered to, then the sampled signal could be reconstructed to its original band limited continuous time form (Stranneby & Walker, 2004).

2.2.2 Aliasing

When sampling a signal which contains frequencies higher than f_{max}, aliasing occurs. The samples will represent information incorrectly from what is actually present in the original. Once aliasing occurs, it is impossible to reconstruct the original data from the sampled data (Proakis & Manolakis, 1996:pg. 27).

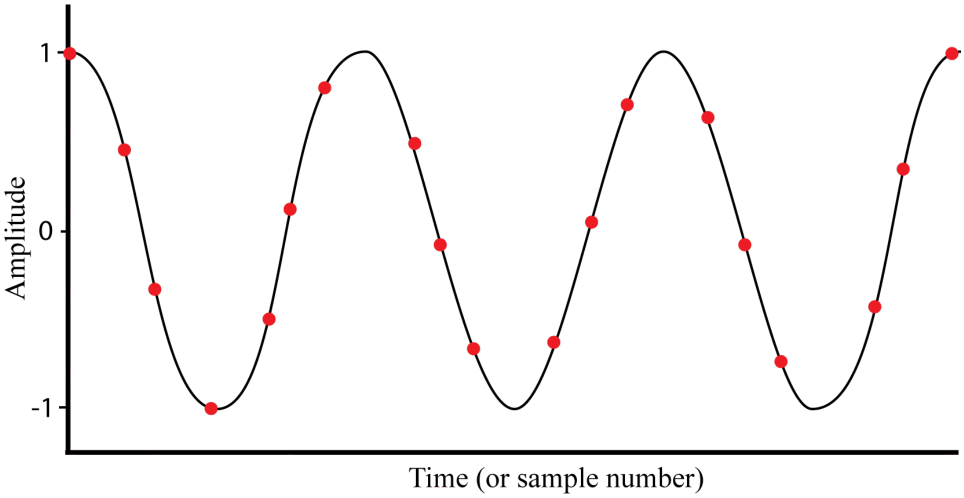


Figure 2.1: Correct sampling which will not result in aliasing adapted from Tan (2008)

Figure 2.1 illustrates correct sampling, while Figure 2.2 does not. The frequency of the sine wave in Figure 2.2 is greater than the Nyquist frequency. This results in aliasing, where the frequency of the sampled signal differs from that of the continuous signal. Aliasing corrupts the data and the original signal cannot be reconstructed from the samples.

In practice, the analogue signal may contain high-frequency noise. In order to satisfy the sampling theorem, a low-pass anti-aliasing filter is applied to limit the input analogue signal, so that all the frequency components are less than f_{max} (Tan, 2008) .

2.2.3 Quantisation

A quantiser is a nonlinear system that transforms a continuous input signal $x(t)$ into a discrete sequence $x[n]$ for which each value assumes a finite number

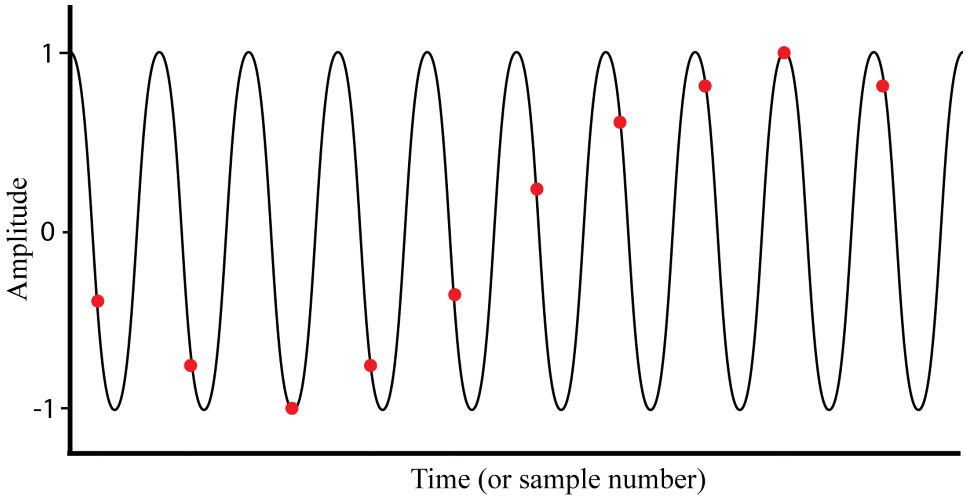


Figure 2.2: Incorrect sampling which leads to aliasing adapted from Tan (2008)

of possible values (Hayes, 1999). This operation is represented by

$$x[n] = Q[x(t)]. \tag{2.2.4}$$

Assigning a numerical value to a signal which is analogue and naturally has an infinite number of values, introduces error. The error being the difference between the rounded sample value and the actual value. With rounding, the quantiser error is

$$e[n] = Q[x(t)] - x(t). \tag{2.2.5}$$

With each additional bit added to the quantiser, the error decreases. According to Hayes (1999), the Signal-to-Quantisation Noise Ratio (SQNR) increases by approximately 6 dB for each bit added. The resulting SQNR yielded from the available bits follows from the equation

$$SQNR \approx 6\ dB \times bit\ length. \tag{2.2.6}$$

2.3 Hardware for signal processing

Many devices are currently available. The leading manufacturers of DSP chips include Analog Devices®, Motorola® and Texas Instruments®. Their products are dedicated to signal processing.

An alternative to using a DSP chip is to use a Field Programmable Gate Array (FPGA) device. Their versatility has recently led to them being used for many different applications, including signal processing. The primary manufacturers used today are Actel®, Altera® and Xilinx®.

The execution speed of most DSP algorithms is limited almost completely by the number of multiplications and additions required. Yet the computational performance must be known and have a predictable execution time. For an application where processing is continuous, or real-time, there is no definite start or end. All processing needs to be completed in an allotted time-frame. An example would be a DSP system for a hearing aid. The designer would need to make sure that if the digital signal processor is receiving 20 kilo-samples per second, it will be able to maintain that throughput of information constantly.

2.3.1 Fixed point and floating point formats

The arithmetic performed on digital hardware can be divided up into two separate categories, fixed point and floating point. They refer to the format used to store and manipulate data within the device. In particular, the formats are used to represent a negative number using binary (Antoniou, 2006:pg. 620).

Fixed point processors represent the data using integer arithmetic. Floating point on the other hand, represents a number using a mantissa and an exponent in addition to the integer format.

Tan (2008:pg. 420) states that since the fixed point digital signal processor operates using an integer format, overflow of data may occur. This leads to much more time spent on coding an error free fixed point based system. Floating point processors offer a much wider dynamic range of data, leading to overflows occurring much less frequently. However, the floating point processor contains more hardware to manipulate the arithmetic, hence is more expensive and slower than a fixed point processor in terms of instruction cycles.

The general format in which a floating point number is given by $x = M \cdot 2^E$, where M is the mantissa, or fractional part and E is the exponent. The exponent and mantissa are both signed numbers. Using a word length of 16 bits, a representation is shown in Figure 2.3. 12 bits are assigned to the mantissa and 4 bits to the exponent, where the normalised range is between -1 and 1. The larger the number of bits assigned to the mantissa, the smaller the interval between values in the normalised range. The larger that the exponent is, the larger the dynamic range becomes.

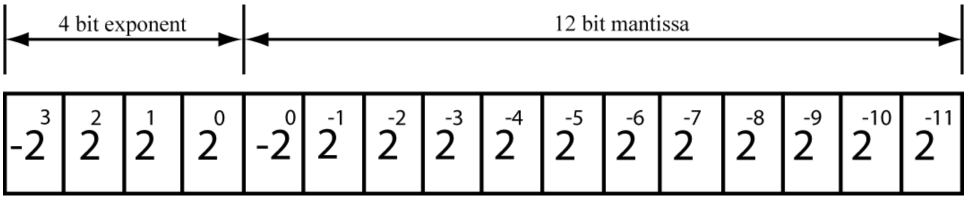


Figure 2.3: Floating point format adapted from Tan (2008)

It is possible to represent negative values using a fixed point processor using the aforementioned 2's complement concept. The most significant bit (MSB) of the word is used to represent the sign of the integer. Considering a 3 bit 2's complement, all the decimal numbers that can be represented are shown in Table 2.1.

Table 2.1: A 3 bit 2's complement number representation

Decimal number	2's complement
3	011
2	010
1	001
0	000
-1	111
-2	110
-3	101
-4	100

A 16 bit unsigned integer can take on any integer value between 0 and 65535. When 2's complement is employed, the range of integers for a 16 bit word is from -32768 to 32767.

It is also possible to have a fractional binary 2's complement system. Tan (2008) states that the Q-format number representation is the most commonly used fractional implementation in fixed point DSP. Illustrated in Figure 2.4 is a Q-15 representation, Q-15 means that the data is in a signed magnitude form where there are 15 bits for magnitude and one bit for the sign. The number is normalised to the fractional range of -1 to 1. The range is divided into 2^{16}

intervals. The most negative number is -1, while the most positive number is $1 - 2^{-15}$.

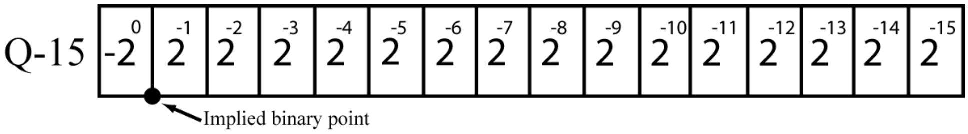


Figure 2.4: Q-15 (fixed point) format adapted from Tan (2008)

2.4 Conclusion

This chapter introduced three elementary signal theory concepts which were sampling, aliasing and quantisation. These ideas are important to note when digital signal processing is to be employed. The audio information presented to the system will be in the digital domain and therefore digital signal processing becomes pertinent in order to achieve the goals set forth in the first chapter.

Hardware devices which could be used for signal processing were introduced as well as fixed point and floating point number systems.

Chapter 3

Pulse width modulation

This chapter describes the concept of Pulse Width Modulation (PWM), presenting a variety of schemes available. Their performance is determined by detailing the harmonic distortion present in the audio band.

Any modulation scheme used for a class-D amplifier aims to create a train of pulses, which when averaged, contains the original reference data (Holmes & Lipo, 2003). One of the problems which face designers is that the pulses also contain unwanted harmonics which should be minimised as much as possible.

PWM is a convenient choice when driving a class-D amplifier, due to the switching characteristics of both.

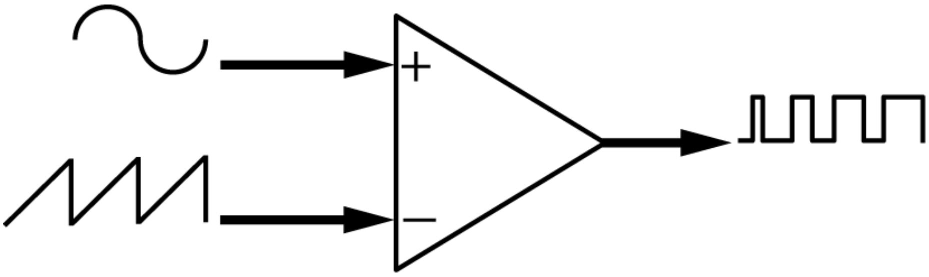


Figure 3.1: Archetype pulse width modulator

3.1 Theory of operation

The PWM signal is generated when two waveforms are compared against each other. The one being a low-frequency reference waveform, audio information in this case, and the other a high-frequency carrier waveform. Figure 3.1 is a diagram of this process. The carrier frequency is usually chosen as being at least ten times the highest frequency in the band of interest (Bresch & Padgett, 1999) .

$$f_{sw} \geq 10 \times f_{max} \tag{3.1.1}$$

When the amplitude of the reference is larger than that of the carrier, the output of the comparator is designated a high value, and a low value when the carrier is larger. The carrier can be either a trailing, leading or a double edged waveform. The resulting PWM waveform differs depending on the carrier used and whether natural sampling, shown in Figure 3.1, or uniform sampling, shown in Figure 3.2, is involved.

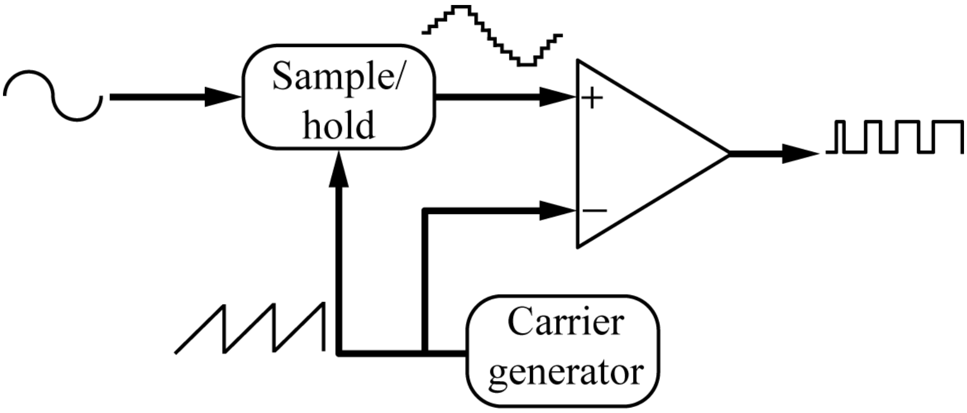


Figure 3.2: Uniformly sampled pulse width modulator adapted from Goldberg & Sandler (1991*a*)

3.2 Natural and uniform sampling

Natural sampling differs from uniform sampling in that it continuously samples the reference and carrier waveforms. Uniform sampling is employed in discrete time systems and is illustrated in Figure 3.2. The difference which results when using either of these two methods is shown in Figure 3.3. The natural reference is an exaggerated version of what it would be in reality, yet it illustrates that the difference between a continuous and sampled signal can cause an error in the resulting PWM waveform. The error in the time difference between the PWM waveforms results in distortion.

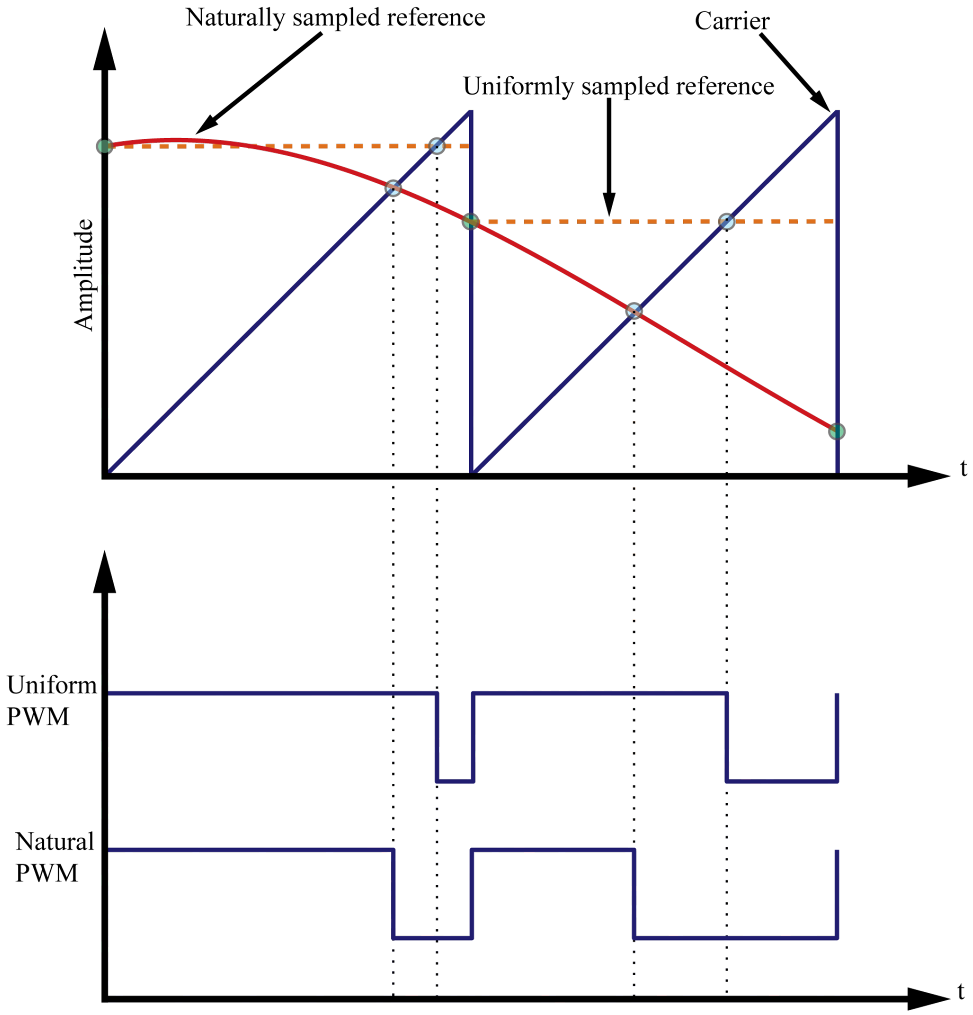


Figure 3.3: UPWM and NPWM producing different PWM waveforms adapted from Nielsen (1998) and Jacobs (2006)

3.3 Harmonic distortion

The spectral contents of a PWM signal with a single frequency sinusoidal input have been determined using a two dimensional Fourier series. The contents of a unity-amplitude trailing edge NPWM for modulation by $M \cos \omega_v t$ into sinusoidal parts yields (Bresch & Padgett, 1999):

$$\begin{aligned} F(t) = k &+ \frac{M}{2}\cos\omega_v t + \sum_{m=1}^{\infty}\frac{\sin m\omega_c t}{m\pi} \\ &- \sum_{m=1}^{\infty}\frac{J_0(m\pi M)}{m\pi}\sin(m\omega_c t - 2m\pi k) \\ &- \sum_{m=1}^{\infty}\sum_{n=\pm 1}^{n=\pm\infty}\frac{J_n(m\pi M)}{m\pi}\sin(m\omega_c t + n\omega_v t - 2m\pi k - \frac{n\pi}{2}), \end{aligned} \tag{3.3.1}$$

where ω_c is the angular fundamental frequency of the PWM carrier, ω_v the angular input signal frequency, k the mean amplitude of the unmodulated carrier and J_n denotes a Bessel function of the first kind. M is the modulation index.

The spectral content includes the input frequency, the sums and differences of the input signal and the carrier and its multiples. There are modulation products of the carrier and input signal which move towards the input signal frequency with decreasing amplitudes. These are referred to as sideband harmonics of the carrier.

The spectral content of a trailing edge UPWM signal yields a rather different result. For the same unity amplitude stimulus as used in Equation 3.3.1 the uniformly sampled PWM spectral content is

$$\begin{aligned} F(t) = k &- \sum_{n=1}^{\infty}\frac{J_n\left(\frac{n\pi M\omega_v}{\omega_c}\right)}{\frac{n\pi\omega_v}{\omega_c}}\sin\left(m\omega_v t - \frac{2n\pi k\omega_v}{\omega_c} - \frac{n\pi}{2}\right) \\ &+ \sum_{m=1}^{\infty}\frac{1 - J_0(m\pi M)}{m\pi}\sin m\omega_c t \\ &- \sum_{m=1}^{\infty}\sum_{n=\pm 1}^{n=\pm\infty}\frac{J_n\left[(m\omega_c + n\omega_v)\right]\frac{\pi M}{\omega_c}}{(m\omega_c + n\omega_v)\frac{\pi}{\omega_c}}\sin\left((m\omega_c + n\omega_v)\left(t - \frac{2\pi k}{\omega_c}\right) - \frac{n\pi}{2}\right). \end{aligned} \tag{3.3.2}$$

The first sum in Equation 3.3.2 shows that the spectral content includes harmonics of the modulating frequency. The amplitudes of the harmonics increase with an increase of modulation index. It also contains sums and differences of the carrier and modulating signal frequency and multiples of the carrier.

3.4 Conclusion

What is understood is that natural pulse width modulation would be preferred over uniform pulse width modulation. This is due to the harmonic content which is present in the baseband of the uniformly sampled PWM. There is however no way to implement NPWM with a digital controller.

The first solution to this problem is to oversample the audio waveform. When more samples are available for a time period the resulting UPWM will be a closer match to NPWM. Unfortunately the audio data has already been sampled at a rate which is much lower than that which would affect linearity. The only solution is to reconstruct a representation of an analogue version of the waveform and extract new data from it. This is accomplished through the use of interpolation, which is presented in Chapter 4. The usefulness of DSP when used with an audio amplifier becomes more apparent.

Chapter 4

Interpolation

4.1 Introduction

The word interpolation has its origins in the Latin word *interpolare*. Which is a combination of *inter*, meaning between and *polare* which means to polish (Meijering, 2002). It is the process of smoothing between the data samples.

Constructing a continuous function becomes a problem when using discrete data which does not contain all of the original information. With so much of the world's data being stored, processed and analysed digitally this is a recurring concern. The easiest and most widely used method for solving this problem is through the use of interpolation. New data is constructed which agrees with the unknown original function (Meijering, 2002).

When a data set is available, interpolation is the method of finding missing data within the confines of the available data. Extrapolation is used when the value of a function outside of the given range is required (Goyal, 2007).

The primary reason for using interpolation in this project is to increase the sample rate of the system. Increasing the sample rate expands the system's frequency bandwidth which then makes it possible to employ certain noise shaping techniques which will be explained in Chapter 5.

This chapter introduces sample rate conversion and its value within the scope of this project. Interpolation is presented as a method of generating new data samples from pre-existing information, which makes sample rate conversion possible. Simulated examples are given to explain the interpolation process.

4.2 Sample rate conversion

Increasing or decreasing the sample rate of a digital system is often required in DSP applications. Crochiere & Rabiner (1981) and Tan (2008:pg. 557) gave such examples as antenna systems, communications, radar systems, audio and speech processing as being multi-rate digital systems.

Downsampling or decimation are terms used when referring to the reduction of the sample rate. While upsampling refers to the increase. The focus of this project is to achieve a higher sampling rate, therefore downsampling will be overlooked. If a fractional sample rate change is required, then a combination of both is necessary.

Crochiere & Rabiner (1981) showed that increasing the sample rate (T_s) by an integer factor of L, would result in a new sampling period of T_s', which is

$$\frac{T_s'}{T_s} = \frac{1}{L}. \tag{4.2.1}$$

This would result in a new sample rate being expressed as $f_s' = Lf_s$.

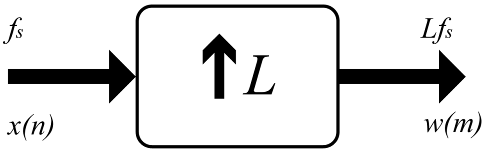


Figure 4.1: Upsampler

Upsampling by a factor L can be achieved by adding $L - 1$ new sample values between each pair of samples. $L - 1$ number of zeros are placed between the data samples which allows for the data to be sampled at a higher rate, as there are now more samples available. Figure 4.1 is a basic block diagram of the process of increasing the sample rate. The input signal $x(n)$ is padded with zeros and then becomes

$$w(m) = \begin{cases} x(m/L), & m = 0, \pm L, \pm 2L, \cdots \\ 0, & otherwise \end{cases} \tag{4.2.2}$$

The output of the sample rate converter $w(n)$ is described in the z-domain by Crochiere & Rabiner (1981) as being

$$\begin{aligned} W(z) &= \sum_{m=-\infty}^{\infty} w(m)z^{-m} \\ &= \sum_{m=-\infty}^{\infty} x(m)z^{-mL} \\ &=X(z^L). \end{aligned} \tag{4.2.3}$$

The upper portion, (a), of Figure 4.2 shows a sine wave which was sampled at 48 kHz. The upsampled version where zeros have been place between subsequent samples is shown in the lower portion, (b), of Figure 4.2. By placing three zero valued samples between each original sample, the sample rate is effectively quadrupled.

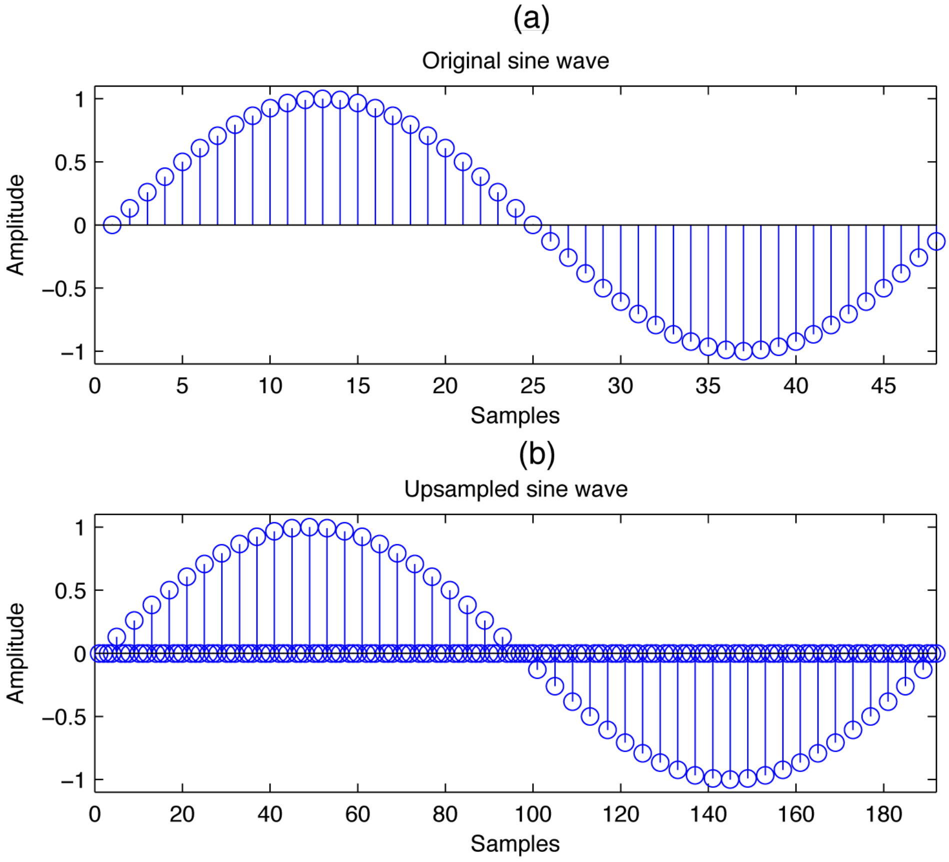


Figure 4.2: Sine wave before and after upsampling

This may seem as though it is the ideal method for achieving any sample rate required. The problem however is evident when viewing the spectral content of the newly upsampled waveform. Figure 4.3 shows the frequency components of Figure 4.2. Aliasing becomes apparent due to the spectral replicas which were originally centred at the sampling frequency, f_s, and its multiples (Tan, 2008). Due to upsampling, the replicas now sit within the newly increased bandwidth.

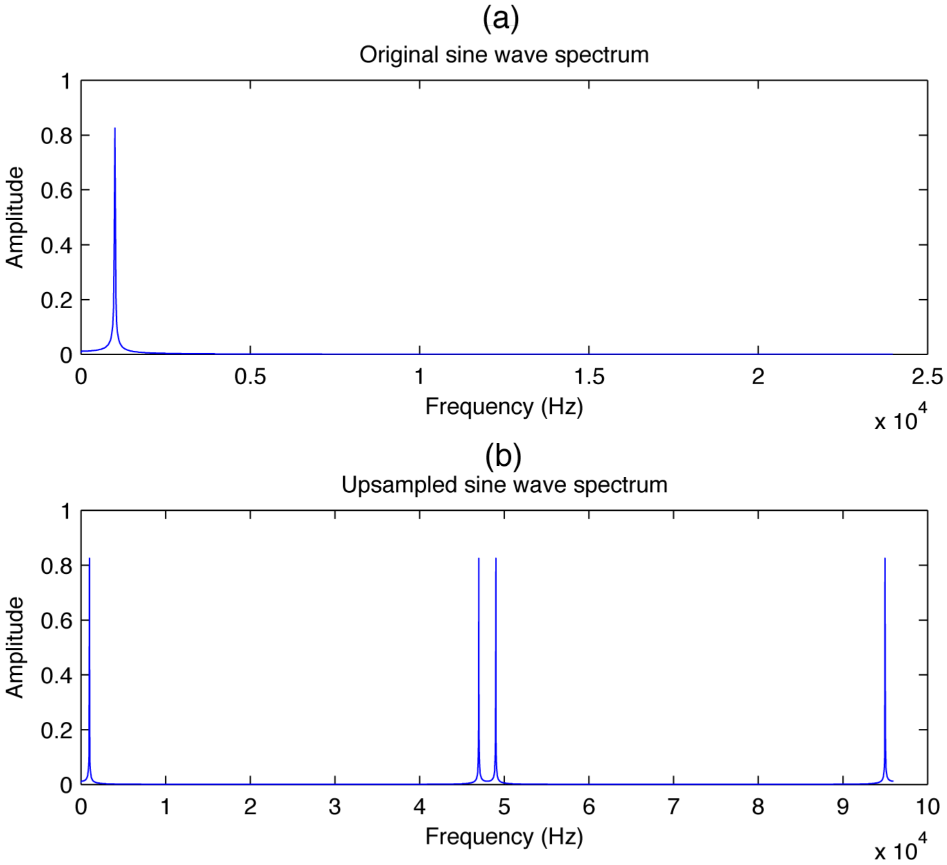


Figure 4.3: 1 kHz sine wave spectrum before and after upsampling

Crochiere & Rabiner (1981) further showed that in order to recover the baseband signal and terminate the unwanted replicas it becomes necessary to filter the signal $w(m)$ with a low-pass filter. Placing the filter in series with the upsampler as shown in Figure 4.4 is necessary. The filter needs to be as close as it can be to the ideal characteristic of

$$H(e^{j\omega'}) = \begin{cases} G, & |\omega'| \leq \frac{2\pi f_s T_s'}{2} = \frac{\pi}{L} \\ 0, & otherwise \end{cases} \tag{4.2.4}$$

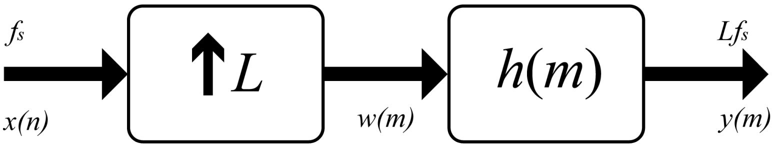


Figure 4.4: Upsampler with low-pass filter

In order for the amplitudes of $x(n)$ and $y(m)$ to match, the gain of the filter G needs to be L in the audio band (Crochiere & Rabiner, 1981). The low-pass filter should have a stop frequency edge of $f_s/2$ (Tan, 2008).

4.3 Finite impulse response interpolation filter

Finite Impulse Response (FIR) as well as Infinite Impulse Response (IIR) filters are the two main categories of filters used for most DSP applications. FIR filters use convolution while IIR filters utilise recursion in their operation. Filters which use convolution outperform those that use recursion, yet they execute at a much slower rate (Antoniou, 2006).

The differences between FIR and IIR filters which were highlighted by Chitode (2009) are detailed in Table 4.1. The FIR filter has an important advantage over the IIR filter, that being a linear phase response in the passband (Jacobs, 2006). It is due to this characteristic that it was the filter type chosen.

Certain specifications need to be determined when filtering an upsampled signal. The passband and stopband need to be chosen so that they agree with the limits of the audio band. The ripple of the filter needs to be specified for how flat the frequency response is in the audio band. Other important considerations are that of the final sampling frequency and the aliasing attenuation. The filter specifications chosen are shown in Table 4.2. The original sampling frequency was 48 kHz which was then upsampled to 192 kHz.

Matlab® was used to design the filter. When the design specifications were placed into Matlab® 's Filter Design and Analysis Tool, a filter length of 334

Table 4.1: Comparison of FIR and IIR filters

Parameter	**FIR**	**IIR**
Recursive/Non-recursive	Do not use feedback	Use feedback hence they are recursive
Phase	Linear phase response	Nonlinear phase response
Stability	Inherently stable	Need to be designed for stability
Number of multiplications	More	Less
Complexity of implementation	Less	More
Required memory	More	Less
Order of filter for similar specifications	High	Low
Design procedure	Less complicated	Complicated
Applications	Used where linear phase is essential	Used where sharp cutoff characteristics with minimum coefficients are required

Table 4.2: FIR filter specifications

Passband	20 kHz
Stopband	24 kHz
Passband ripple	0.001 dB
Stopband attenuation	144 dB
Filter sampling frequency	192 kHz

points was generated. The magnitude response of the FIR filter is shown in Figure 4.5. It can be clearly seen that the images above 24 kHz have been attenuated.

Viewing the phase response of the FIR shown in Figure 4.5 confirms the linear nature which it should have. Note that the phase response is only linear in the passband. The upsampled waveform shown in Figure 4.2 was then applied to the filter. The resulting spectral contents are shown in Figure 4.6.

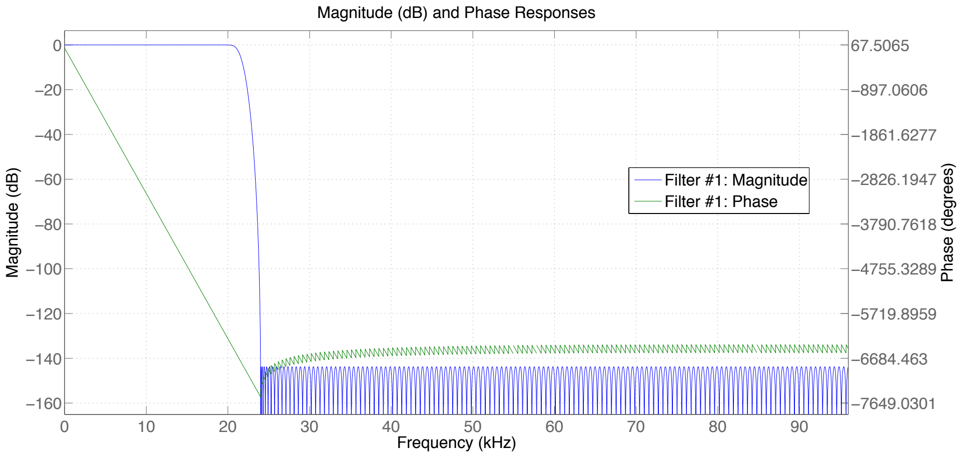


Figure 4.5: FIR filter magnitude and phase response

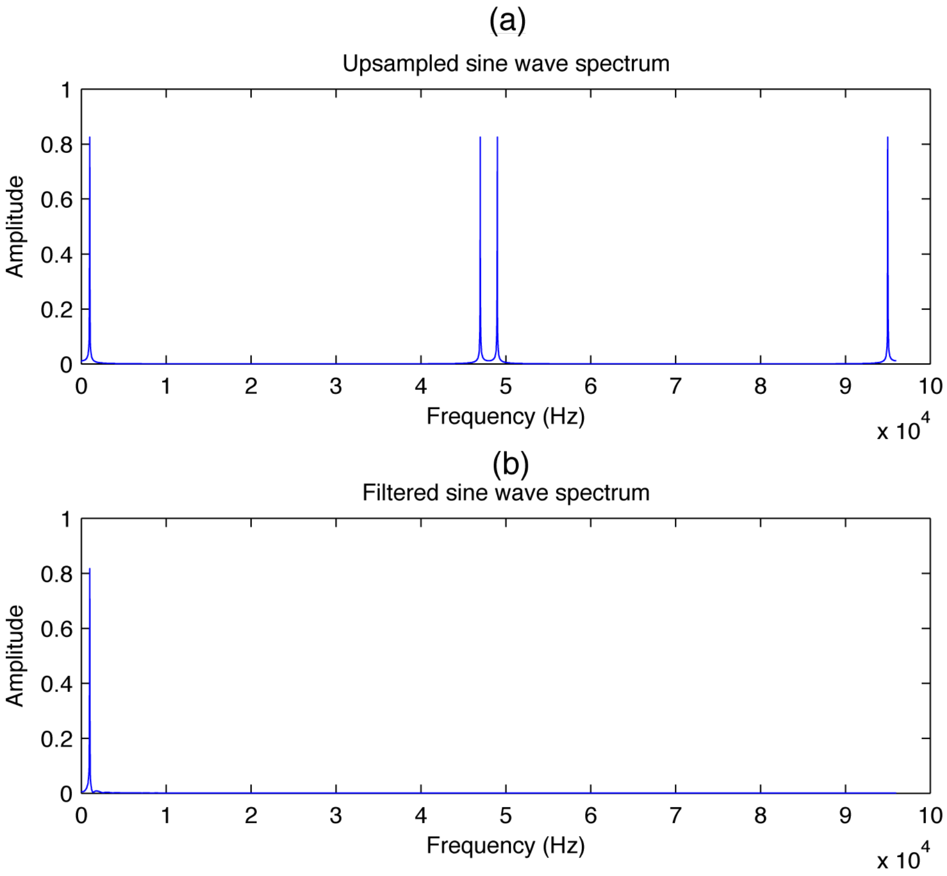


Figure 4.6: 1 kHz sine wave spectrum before and after being filtered

4.4 Polyphase interpolation filter

Due to the characteristics of the interpolation process, the polyphase filter structure can be used to efficiently implement the interpolation filter. It uses fewer multiplications and additions to complete the same task as that of a regular FIR described in Section 4.3 (Tan, 2008).

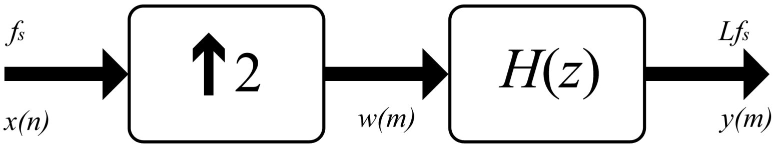


Figure 4.7: Direct interpolation filter

It is an all pass filter with different phase shifts, hence the name polyphase filter. Figure 4.7 shows an interpolation process where $L = 2$. Assuming that the FIR filter has four filter coefficients, shown as

$$H(z) = h(0) + h(1)z^{-1} + h(2)z^{-2} + h(3)z^{-3}. \tag{4.4.1}$$

The output of the filter would then be

$$y(m) = h(0)w(m) + h(1)w(m-1) + h(2)w(m-2) + h(3)w(m-3). \tag{4.4.2}$$

Tan (2008) showed that with the configuration in Figure 4.7 that eight multiplications and six additions would be required. Table 4.3 shows the results of the direct interpolation of Figure 4.7, where $w(m)$ is the upsampled signal and $y(m)$ is the interpolated output.

Table 4.3: Result of the direct interpolation process

n	$x(\mathrm{n})$	m	$w(m)$	$y(\mathrm{m})$
n = 0	$x(0)$	$m = 0$	$w(0) = x(0)$	$y(0) = h(0)x(0)$
		$m = 1$	$w(1) = 0$	$y(1) = h(1)x(0)$
n = 1	$x(1)$	$m = 2$	$w(2) = x(0)$	$y(0) = h(0)x(1)+h(2)x(0)$
		m = 3	$w(3) = 0$	$y(0) = h(1)x(1)+h(3)x(0)$
n = 2	$x(2)$	$m = 4$	$w(4) = x(0)$	$y(0) = h(0)x(2)+h(2)x(1)$
		$m = 5$	$w(5) = 0$	$y(5) = h(1)x(2)+h(3)x(1)$
...	...	...	...	...

The same results shown in Table 4.3 can be calculated using a polyphase structure. The benefit would be that only four multiplications and four additions are required. The polyphase structure for this example is shown in Figure 4.8.

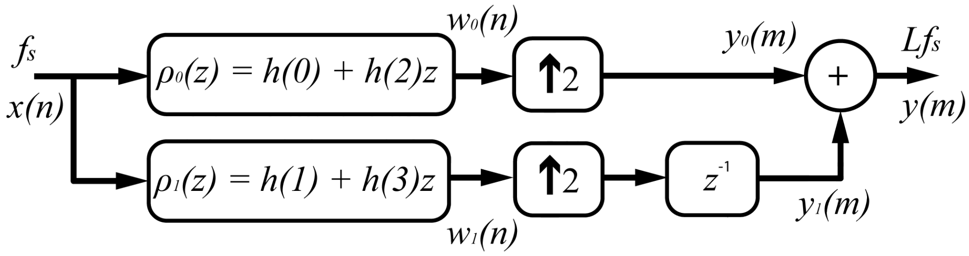


Figure 4.8: Polyphase filter implementation adapted from Tan (2008)

The original FIR filter is split up into L polyphase filters. This characteristic eliminates the need to add zeros between samples (Jacobs, 2006). Once the original filter has been designed containing N coefficients, Tan (2008:pg. 584) showed that the coefficients for the sub filters can be determined according to

$$\rho_k(n) = h(k + nL) \text{ for } k = 0, 1, ..., L - 1 \text{ and } n = 0, 1, ..., \frac{N}{L} - 1. \quad (4.4.3)$$

The computational cost can be reduced by a factor of L when compared against the direct interpolation method shown previously. Figure 4.9 shows the commutative model for the polyphase filter. The sample rate increase and delays are replaced by a commutator which alternates between the different branches at a rate of Lf_s.

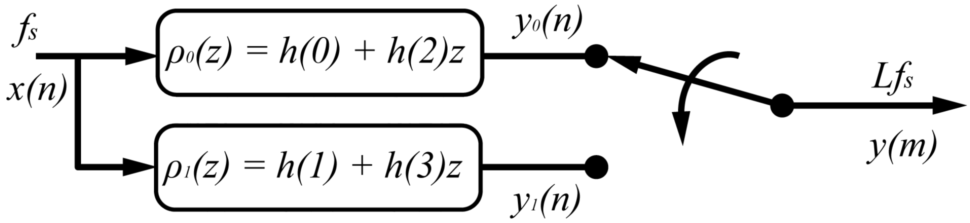


Figure 4.9: Commutative model for the polyphase interpolation filter adapted from Tan (2008)

This is where the efficiency of the polyphase filter becomes apparent. The filtering is performed at the original sampling rate of f_s which is much lower than Lf_s (Jacobs, 2006). Direct FIR interpolation requires the sample rate of the filtering to be at the higher sample rate of Lf_s.

The impulse response of the original FIR filter is shown in Figure 4.10 while the impulse responses of the two polyphase filters is shown in Figure 4.11 and Figure 4.12.

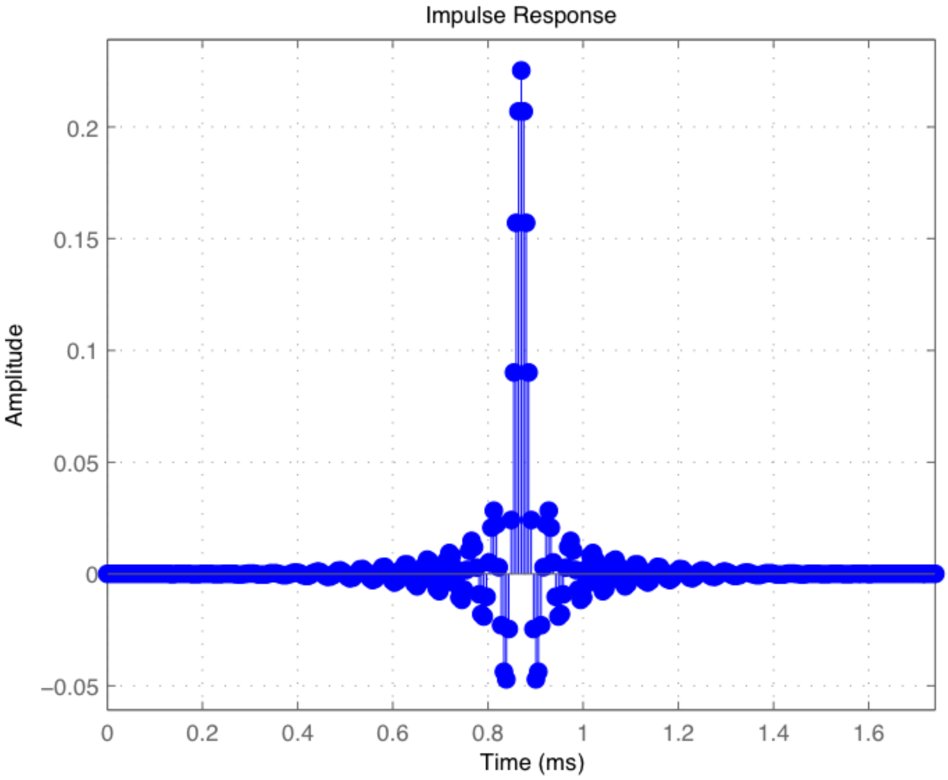


Figure 4.10: Impulse response of the original FIR filter

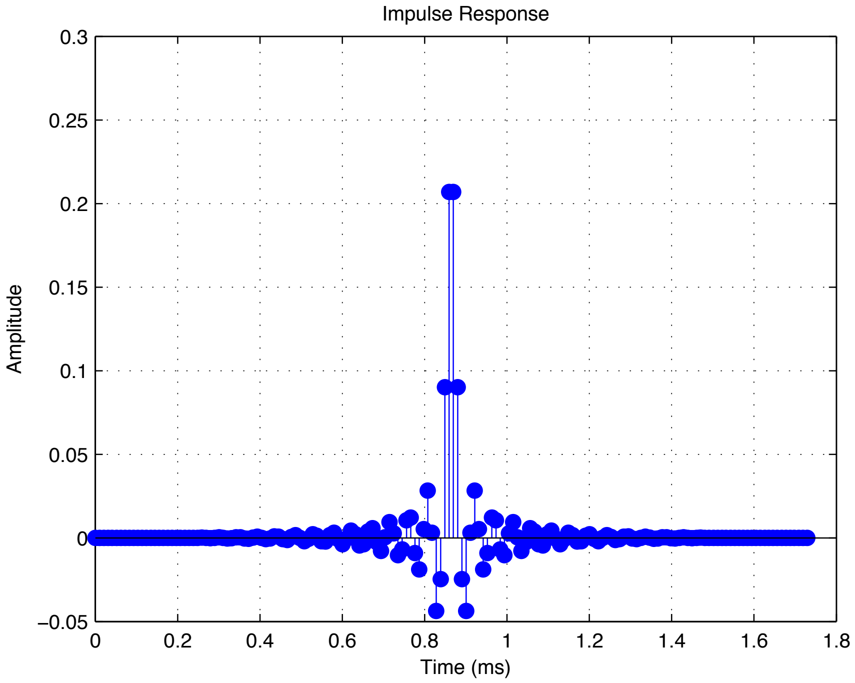


Figure 4.11: Impulse response of the first polyphase filter

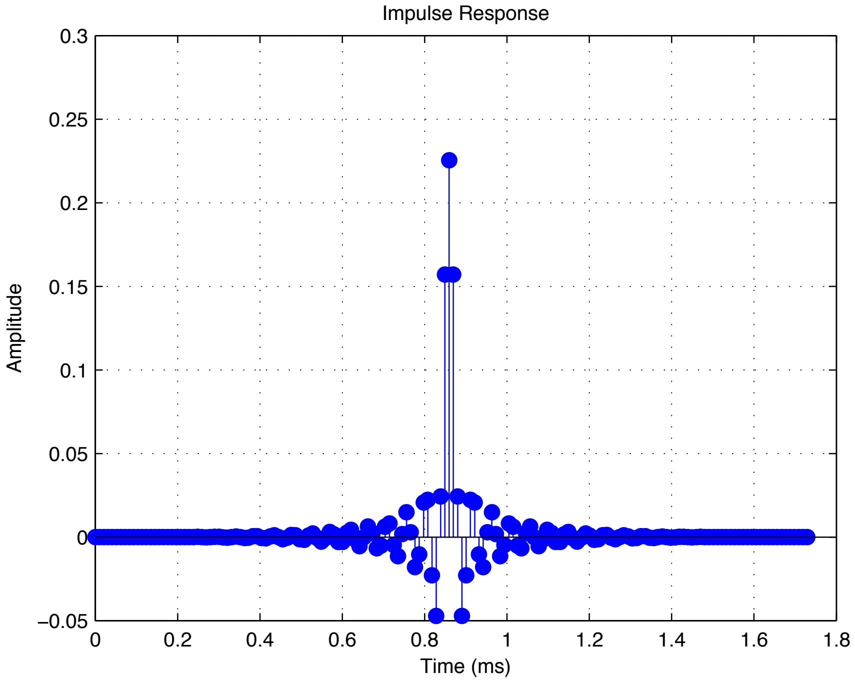


Figure 4.12: Impulse response of the second polyphase filter

4.5 Cascaded integrator comb filter

Hogenauer (1981) developed the Cascaded Integrator Comb (CIC) filter for multi-rate processing. It is a digital linear phase FIR filter specifically aimed at decimation and interpolation as it has a low-pass characteristic. The primary benefits of using this filter type are:

- No multipliers are required,
- no filter coefficients need to be stored in memory,
- the structure is very consistent and made up of only two building blocks,
- no complicated timing is required and
- the same filter can be used for many different sample rate change ratios with minimal adjustments required.

There are however limiting factors when using this filter type, they include:

- Register word lengths can become large for large sample rate changes. The register word length increases exponentially for each additional stage added to the filter.
- The filter's characteristics and frequency response is limited due to being influenced by only three parameters, the sample rate change ratio, L, the differential delay, M, and the number of integrator-comb pairs, N.

The advantages and disadvantages make this filter ideal for systems requiring large rate changes while utilising minimal resources.

The filter consists of integrator-comb pairs which produce a consistent FIR. When interpolation is required, the integrators operate at the new desired sample rate, while the comb section operates at the slower original sample rate. This is another benefit of the filter as half the processing is conducted using a slower clock. The number of integrator-comb pairs is chosen in accordance with the requirements for image attenuation.

The interpolating CIC filter structure is shown in Figure 4.13.

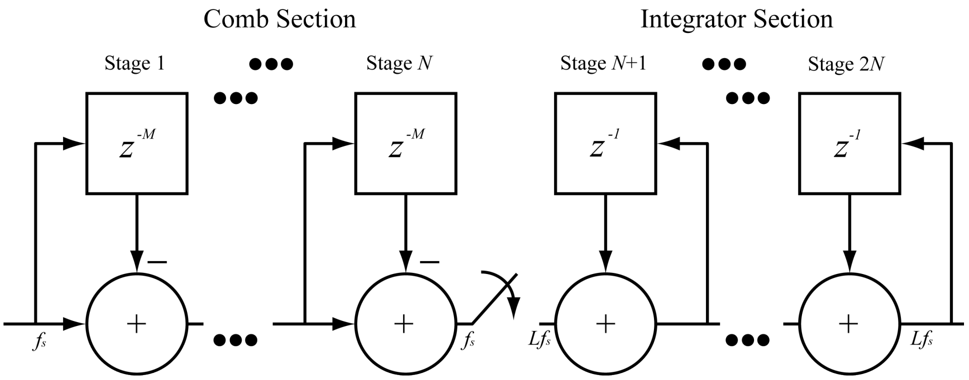


Figure 4.13: CIC filter block diagram adapted from Hogenauer (1981)

The transfer function for each integrator which operates at a sampling rate of Lf_s, where L is the oversampling ratio, is

$$H_I(z) = \frac{1}{1 - z^{-1}}. \tag{4.5.1}$$

The comb section operates at the original sampling rate of f_s. The differential delay, M, is a design parameter used to control the frequency response of the filter. The transfer function for each comb section is

$$H_C(z) = 1 - z^{-LM}. \tag{4.5.2}$$

The rate change occurs by placing $L - 1$ zero value samples between the original data samples coming out of the comb section. Combining Equations 4.5.1 and 4.5.2 yields a transfer function for the CIC filter of

$$H(z) = H_I^N(z) H_C^N(z) = \frac{(1 - z^{-LM})^N}{(1 - z^{-1})^N} = \left[\sum_{k=0}^{LM-1} z^{-k} \right]^N. \tag{4.5.3}$$

When designing a CIC interpolator, attention needs to be paid to the bit growth of the filter sections. The introduction of a small error into the integrator stages, resulting from rounding, would cause the error difference to increase until the filter becomes unstable. Therefore the maximum word length for each stage needs to be known and accommodated for. Both Frerking (1994) and Hogenauer (1981) mentioned the importance of not truncating the word lengths within the integrator stages.

The maximum word length increase up to the j^{th} stage can be shown to be

$$G_j = \begin{cases} 2^j & j = 1, 2, \cdots, N \\ \frac{2^{2N-j}(LM)^{j-N}}{L} & j = N+1, \cdots, 2N \end{cases} \tag{4.5.4}$$

The minimum word length required, based on the growth G_j, where B_{in} is the input word length, is

$$W_j = B_{in} + log_2 G_j. \tag{4.5.5}$$

Truncation can only be employed after the last integrator stage. This becomes the only error introduced. The number of LSBs which can be discarded, where B_{out} is the output word length, is

$$B_T = W_{2N} - B_{out}. \tag{4.5.6}$$

Matlab® provides a filter design tool to help with creating a CIC filter. Even though the basic building blocks will always remain the same, the amount of stages required is easily determined when using the filter design tool. As an example, the following characteristics can be specified and designed in Matlab®:

- Differential delay = 1,
- interpolation ratio = 16,
- pass band frequency = 22 kHz,
- desired sampling rate = 3.072 MHz and
- the aliasing attenuation = 60 dB.

The resulting filter has four sections with a magnitude response shown in Figure 4.14. This filter will upsample a 192 kHz waveform to 3.072 MHz using only a minimal amount of resources. It is evident from Figure 4.14 that truncation of the output is necessary as the gain of the filter is very large.

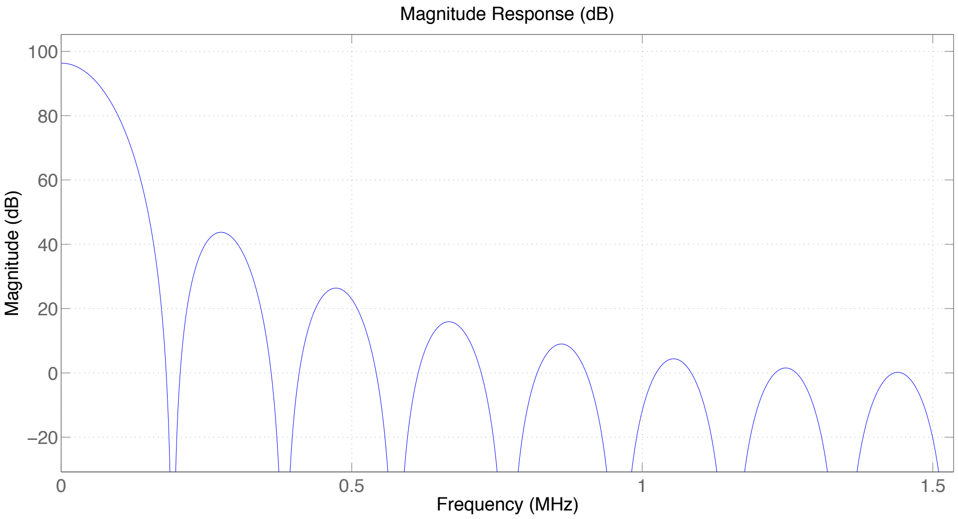


Figure 4.14: CIC filter magnitude response

4.5.1 CIC compensation filter

An unfortunate characteristic of the CIC filter is its non-flat frequency response. There is a way to obtain the benefits gained from using a CIC filter while still maintaining a good frequency response. Cascading a conventional FIR filter in conjunction with the CIC filter allows for the frequency response to be modified to compensate for imperfections. Figure 4.15 shows the configuration which would achieve this.

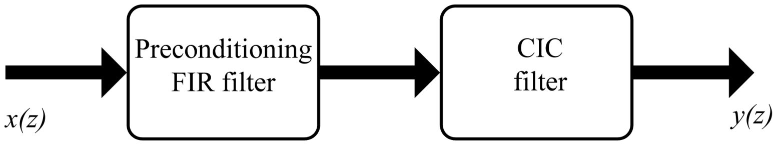


Figure 4.15: CIC compensation filter block diagram

Matlab® can again be used to design the compensation filter as it has specific functionality for this purpose. By entering the number of stages which the CIC filter requires, the filter design tool will generate an FIR filter to help counteract the habitual CIC shape. Figure 4.16 shows the magnitude response of such a filter.

When both filters are cascaded, as shown in Figure 4.15, the magnitude response becomes that shown in Figure 4.17. The cutoff is now much sharper.

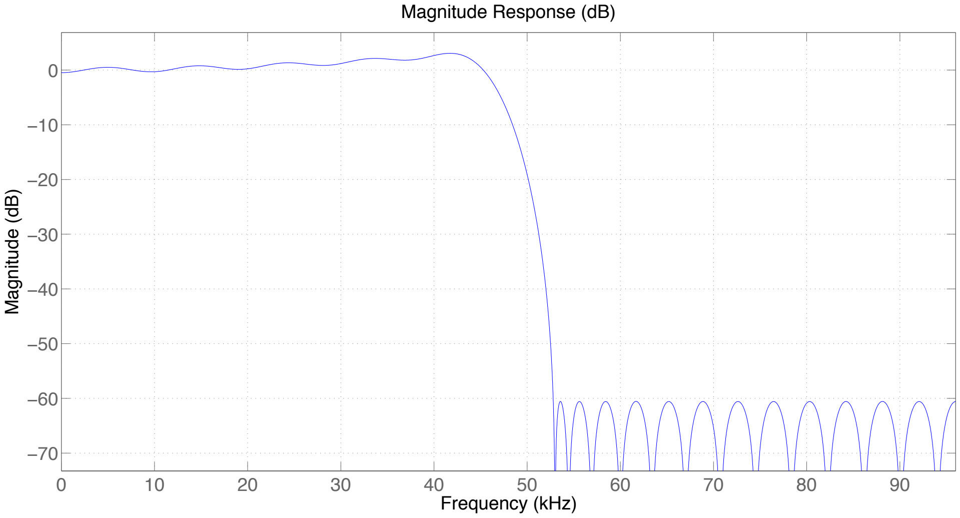


Figure 4.16: CIC compensator filter magnitude response

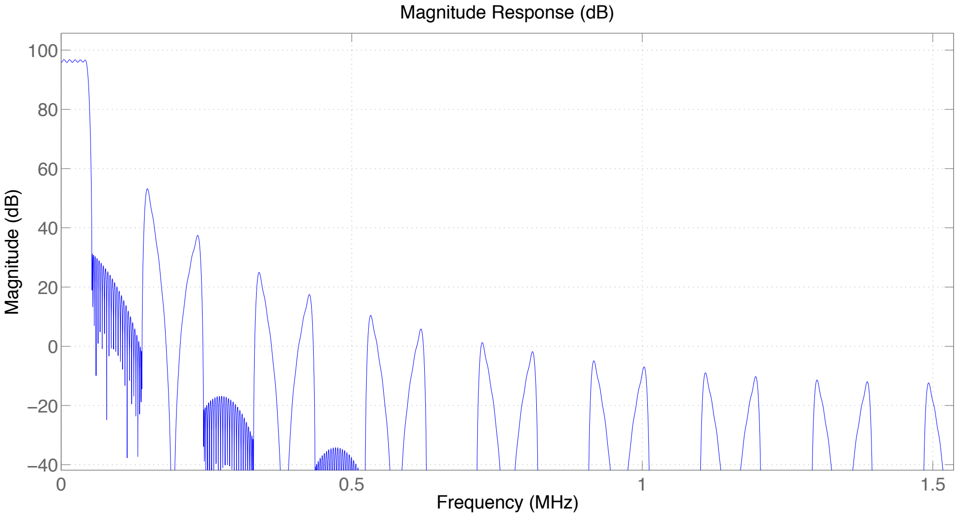


Figure 4.17: Magnitude response of a CIC filter cascaded with a CIC compensator

Depending on the FIR filter used, a better response could be achieved. If resources are vital, which would be the reason for not using a FIR filter in the first place, a half-band FIR filter could be employed. Half-band filters have every second coefficient reduced to zero, making them computationally inexpensive (Lutovac *et al.*, 2001). The performance of a normal FIR filter is achieved while only paying the computational price of half the multiplications.

Lutovac *et al.* (2001) further mentioned that a limiting factor when using the half-band filter is that the frequency symmetry condition must be met, which is

$$f_{stopband} = \frac{1}{2} - f_{passband}. \tag{4.5.7}$$

This means that the design boundaries are constrained. The transition region is centred at a quarter of the sampling rate.

4.6 Polynomial interpolation

This method of interpolation involves constructing a curve through a specific set of data points. Trefethen (2000), Cohen *et al.* (2001), Stetter (2004), Kiusalaas (2005), Goyal (2007), Lyons (2007) and Butt (2008) have presented polynomial interpolation using many different methods.

Constructing a polynomial of degree $n-1$ which will pass through n distinct data points is always possible (Kiusalaas, 2005). The concept is that the polynomial would follow the natural path that a continuous signal would follow through the data points.

If constructed correctly, any new data point could be found along the polynomial, thereby generating as much new data as is required. This idea is illustrated in Figure 4.18 where a fifth order polynomial is constructed passing through six data points. The methods for approximating the polynomial and generating new data from it vary. The Lagrange method as well as Neville's formula and Newton's interpolation formula are presented in the following sections.

4.6.1 Lagrange interpolation

Given the data points $x_0, x_1...x_n$ where $f(x_0), f(x_1)...f(x_n)$ are their corresponding magnitudes, the point located at x can be found through an uncomplicated calculation. Goyal (2007) derived Lagrange's interpolation formula to

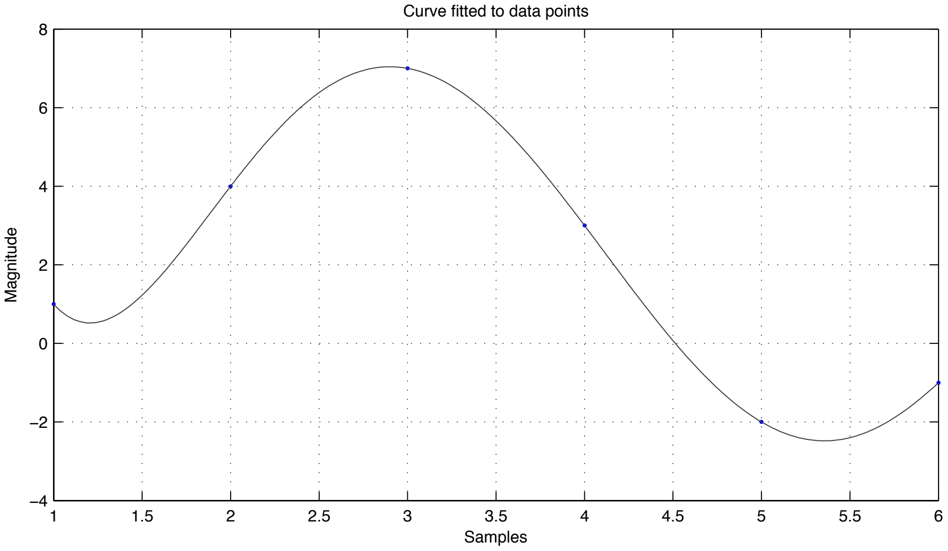


Figure 4.18: Curve constructed from discrete data points

be

$$
\begin{aligned}
f(x) = & \frac{(x-x_1)(x-x_2)...(x-x_n)}{(x_0-x_1)(x_0-x_2)...(x_0-x_n)} f(x_0) \\
& + \frac{(x-x_0)(x-x_2)...(x-x_n)}{(x_1-x_0)(x_1-x_2)} f(x_1) \\
& + ... + \frac{(x-x_0)(x-x_1)...(x-x_{n-1})}{(x_n-x_0)(x_n-x_1)...(x_n-x_{n-1})} f(x_n).
\end{aligned} \tag{4.6.1}
$$

Lyons (2007) showed that the Lagrange formula could also be represented as

$$f(x) = \sum_{i=0}^{n} f(x_i) \prod_{j=0, j\neq i}^{n} \frac{x-x_j}{x_i-x_j}. \tag{4.6.2}$$

Lagrange's method is simple in theory, yet it is not an efficient algorithm. For each new data point required, a substantial amount of calculation is required. When implemented on embedded hardware the amount of multipliers and time required would be unacceptable. This method is accurate and would be a good choice if only one new data sample was needed.

4.6.2 Newton's interpolation formula

This method of interpolation gains efficiency by calculating the polynomial coefficients and new data values in two separate calculation steps. This means that once the coefficients have been calculated, there can be any number of new data points generated without recalculating the coefficients.

In order to calculate the coefficients for the polynomial, a divided differences method can be employed. Determining the values of a divided differences table can be completed by using the formula

$$\begin{aligned}\Delta^n f_i =& f_{i+n} - n f_{i+n-1} + \frac{n(n-1)}{2!} f_{i+n-2} - \frac{n(n-1)(n-2)}{3!} f_{i+n-3} \\ &+ ... + (-1)^n f_i.\end{aligned} \tag{4.6.3}$$

Creating a divided differences table helps to calculate and collate the coefficients. Table 4.4 shows how to set out a divided differences table when calculating a 2^{nd} order polynomial.

Table 4.4: 2^{nd} order polynomial divided differences

x	$f(x)$	$\Delta f(x)$	$\Delta^2 f(x)$
x_0	y_0		
		$\frac{y_1-y_0}{x_1-x_0}$	
x_1	y_1		$\frac{\Delta f_0-\Delta f_1}{x_2-x_0}$
		$\frac{y_2-y_1}{x_2-x_1}$	
x_2	y_2		

From Table 4.4 the coefficients are taken as the top value of each row, except the first row which contains the data position. Therefore the coefficients for a 2^{nd} order polynomial would become

$$\begin{aligned} a_0 =& \quad y_0 \\ a_1 =& \quad \tfrac{y_1-y_0}{x_1-x_0} \\ a_3 =& \quad \tfrac{\Delta f_0-\Delta f_1}{x_2-x_0} \end{aligned}$$

Once the coefficients have been calculated then any new data point can be generated. When n-data points are used, an $n-1$ order polynomial is possible. The equation used to determine a desired value at point x is

$$\begin{aligned} f(x) =& a_0 + (x - x_0)a_1 + (x - x_0)(x - x_1)a_2 \\ &+ ... + (x - x_0)(x - x_1)...(x - x_{n-1})a_n. \end{aligned} \tag{4.6.4}$$

This equation may appear to be just as computationally expensive as the Lagrange method, yet it is possible to pre-calculate most of the equation offline and store it in a lookup table. If the same amount of new data values at the same points need to be generated each time the coefficients are updated then Equation 4.6.4 can be simplified to

$$f(x) = a_0 + v_1a_1 + v_2a_2 + ... + v_na_n. \tag{4.6.5}$$

Equation 4.6.5 drastically reduces the amount of multipliers required for each iteration. The values of $v_1...v_n$ would be the information stored in a lookup table.

4.6.3 Neville's formula

Neville's formula is based on a similar principle to Newton's interpolation formula (Kiusalaas, 2005). It forms a triangular shaped set of results, appearing similar to that of a divided differences table. The formula is recursive and each iteration fits a polynomial of a degree higher than the previous iteration. The number of iterations relates to the order of the polynomial required.

If $P_k[x_j, x_{j+1}, ..., x_{j+k}]$ represents a polynomial of degree k that passes through $k + 1$ data points. Then the general recursive formula as presented by Kiusalaas (2005) is

$$\begin{aligned} &P_k[x_i, x_{i+1}, ..., x_{i+k}] \\ &\quad = \frac{(x - x_{i+k})P_{k-1}[x_i, x_{i+1}, ..., x_{i+k-1}] + (x_i - x)P_{k-1}[x_{i+1}, x_{i+2}, ..., x_{i+k}]}{x_i - x_{i+k}}. \end{aligned} \tag{4.6.6}$$

When searching for the value of x, the calculations can be conducted using a table as shown in Table 4.5.

Table 4.5: 2^{nd} order polynomial using Neville's algorithm

	k=0	k=1	k=2
x_0	$P_0[x_0] = y_0$		
		$P_1[x_0, x_1]$	
x_1	$P_0[x_1] = y_1$		$P_2[x_0, x_1, x_2]$
		$P_1[x_1, x_2]$	
x_2	$P_0[x_2] = y_2$		

This process results in generating $P_2[x_0, x_1, x_2]$, which is the desired value at point x.

4.6.4 Polynomial interpolation limitations

It is best to carry out the polynomial calculation using the least amount of data points possible. There do however need to be enough points used in order to achieve an accurate representation. Three to six data points provide the most stable result (Kiusalaas, 2005). The instability which can result from too many data points being employed is shown in Figure 4.19. The error occurred at the beginning and at the end of the polynomial calculation.

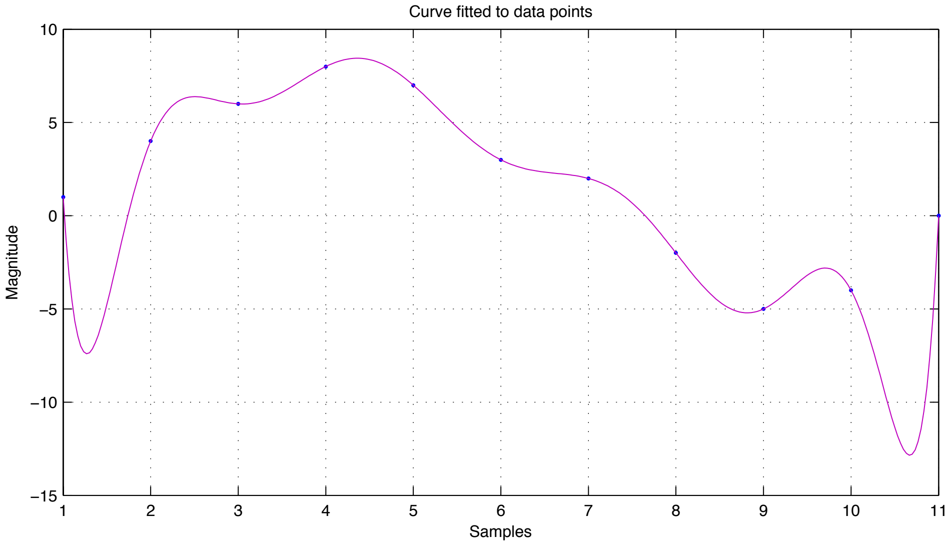


Figure 4.19: Unstable curve constructed from 11 data points

4.7 Conclusion

Directly upsampling a signal to a higher rate was shown to be possible. The resulting waveform was however unacceptable as aliasing occurred in the baseband spectrum. Adding an anti-imaging filter in conjunction with the upsampler proved to be an effective solution. The FIR filter is however resource hungry and once higher oversampling ratios are required, its applicability becomes questionable.

The polyphase structure did help with reducing the required resources necessary for implementation and would be the only practical choice when implementing a large FIR filter on embedded hardware.

When minimal resources are available, the CIC filter proved to be a very valuable method for increasing the sample rate, especially by large amounts. There are undesirable characteristics of the CIC filter which need to be accepted, it has a very gentle roll-off and the gain is very large. Incorporating an additional FIR filter to compensate for the edgeless cutoff was shown to be possible.

Many different methods for polynomial interpolation are available. The Lagrange method, while easier to implement, is not very useful when many new data samples need to be generated successively. The method which showed itself to offer the most benefit for this particular project was Newton's interpolation formula. The calculation of the coefficients is not very complicated and once the coefficients are available, any number of data points can be quickly generated using only a few multiplications and additions.

Interpolation is a useful tool which can be used to increase the amount of samples available. The increase in samples allows for an increase in the sample rate which then leads to a larger frequency bandwidth. With a larger bandwidth there is more freedom to utilise the available frequencies above the audio band. Noise shaping, which is presented in Chapter 5, utilises the unused bandwidth which the interpolation scheme makes available.

Chapter 5

Noise shaping

This chapter presents the concept of noise shaping and its use in signal processing. The benefits gained from using it in combination with a PWM amplifier are explained. Examples are given showing how it improves the performance of a pulse width modulator.

5.1 Introduction

Chapter 4 introduced oversampling and interpolation which can be used to increase the bandwidth from the original smaller audio band. This allows for the use of techniques which reduce the in-band quantisation noise by moving it out of the audio band and up to higher frequencies. It is possible to do this as there are no useful frequency components outside of the audio band (Hawksford, 1989*b*). This technique is referred to as noise shaping.

By reducing the noise power present in the audio band, it is possible to achieve high-resolution audio while still utilising a modest bit rate for the modulator. The resolution of each data sample is reduced and the objective is to locate the extra quantisation noise in the superfluous space created by oversampling, yet at the same time maintaining a minimal amount of distortion in the baseband (Hawksford, 1989*b*). The noise shaping characteristic is that of a high-pass filter. The difference between using noise shaping and not is shown in Figure 5.1. The solid line clearly illustrates the high-pass shape, while the dashed line shows the noise power without noise shaping.

Noise shaping does not eliminate noise from the system, it merely moves it around. It actually increases the overall noise power at higher frequencies when used in an audio application (Kozak & Kale, 2003). The final result is always beneficial as the low pass filter of a class-D power stage attenuates the high frequency components.

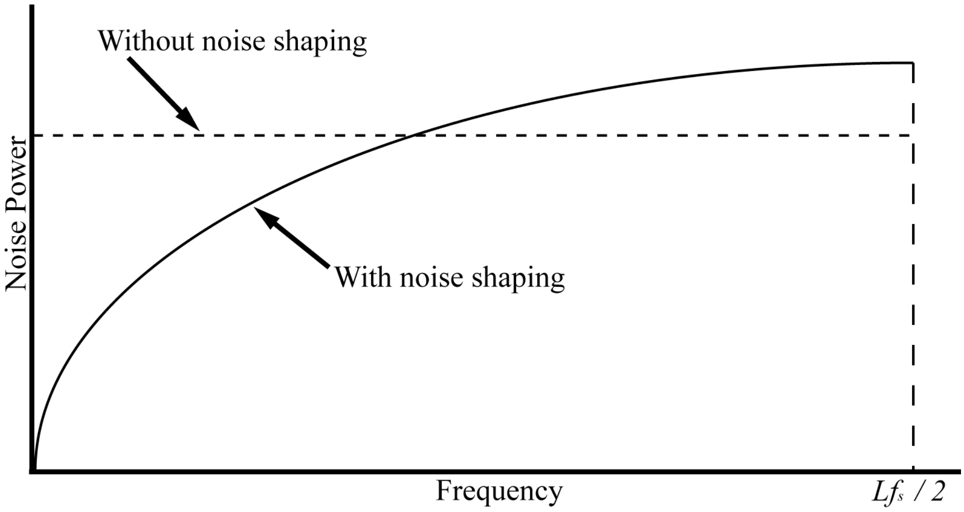


Figure 5.1: Shaped and unshaped noise adapted from Craven (1993)

It was shown by Hawksford (1989*b*) and Logan & Hawksford (1994) that the usual method for obtaining noise shaping is to enclose the pulse width modulator within a local recursive loop. Negative feedback has a distortion shaping characteristic which is then used. Therefore it is possible to use a low resolution signal instead of a high resolution one if the design can operate effectively once oversampled. There can be a trade off between sampling rate and resolution. The higher the sample rate, the lower the resolution. The quantiser can be reduced down to 1-bit, as is the case for delta-sigma ($\Delta\Sigma$) data converters, which use very high sample rates.

5.2 General noise shaping

Shown in Figure 5.2 is a typical noise shaper configuration. It is an entirely digital system where a high resolution input of b-bits is truncated to b'-bits. The quantisation error is added when the least significant bits are removed (Goldberg & Sandler, 1991*a*). The number of bits removed relates to the resolution of the PWM process. The quantisation error is filtered by $H(z)$ and through negative feedback is subtracted from the input.

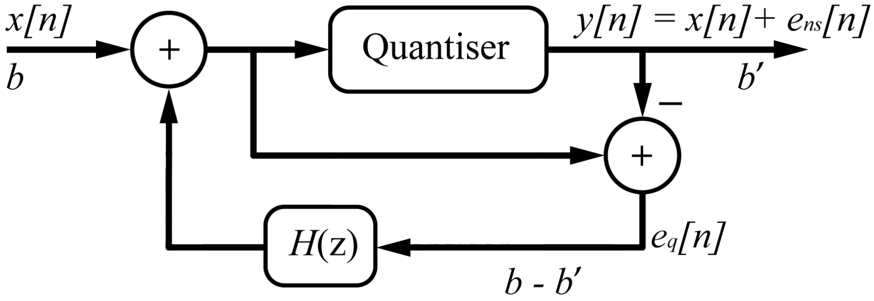


Figure 5.2: Noise shaper configuration adapted from Goldberg & Sandler (1991*a*)

Goldberg & Sandler (1991*a*) provided an analysis of the noise shaper configuration shown in Figure 5.2. $x[n]$ is the high resolution input and $y[n]$ is the coarsely quantised output. The error at the output is represented as $e_{ns}[n]$ which is the noise shaped error. The error which is generated by the quantisation is represented as $e_q[n]$. The z-transforms for these components are $X(z)$, $Y(z)$, $E_{ns}(z)$ and $E_q(z)$ respectively.

If the quantiser is modelled as an additive noise source the output then becomes

$$y[n] = x[n] + e_{ns}[n]. \tag{5.2.1}$$

The quantisation error can be represented in the z-domain as

$$E_q(z) = X(z) + E_q(z)H(z) - [X(z) + E_{ns}(z)]. \tag{5.2.2}$$

The noise transfer function (NTF) can then be constructed as being

$$NTF(z) = \frac{E_{ns}(z)}{E_q(z)} = H(z) - 1. \tag{5.2.3}$$

Tewksbury & Hallock (1978) found that the optimum form for an oversampled noise shaping filter is

$$H(z) = (1 - z^{-1})^N, \tag{5.2.4}$$

where N is the order of the filter.

Shown in Figure 5.3 are the magnitude responses when different orders of Equation 5.2.4 are used. The filter is sampled at 384 kHz, f_s, which is the same as the PWM switching frequency. Jacobs (2006) explained that the magnitude of the transfer function shown in Figure 5.3 will always cut the

0 dB point at $f_s/6$. Therefore a reduction of the quantisation error is only achieved at frequencies less than $f_s/6$.

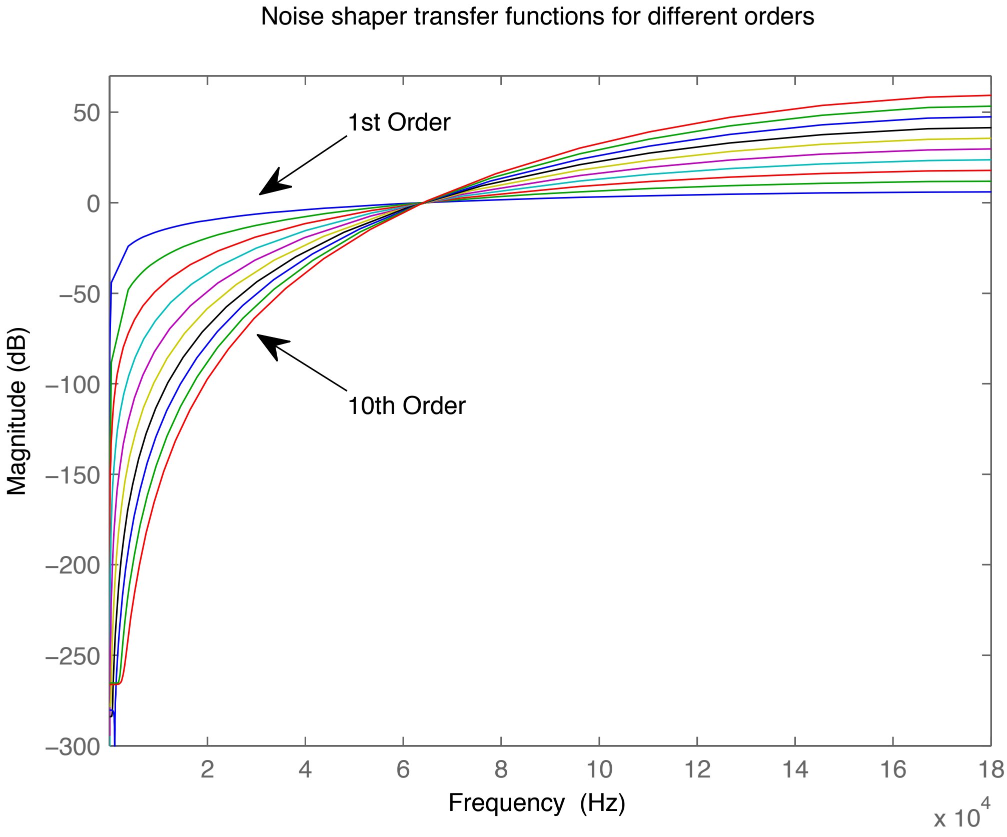


Figure 5.3: Noise shaping filter when calculated for different orders adapted from Hawksford (1989*a*) and Jacobs (2006)

In order to evaluate the noise shaping function of the configuration first shown in Figure 5.2, filter coefficients need to be determined. Equation 5.2.4 is used as the basis for this procedure. Section A.1 in the Appendices details the calculations needed to find the coefficients for the filter. The coefficients were calculated for filter orders of one to five.

The resulting coefficients for a fifth order filter are

$$
\begin{aligned}
a_0 &= -1 \\
a_1 &= 5 \\
a_3 &= -10 \\
a_4 &= 10 \\
a_5 &= -5 \\
a_6 &= 1
\end{aligned}
\tag{5.2.5}
$$

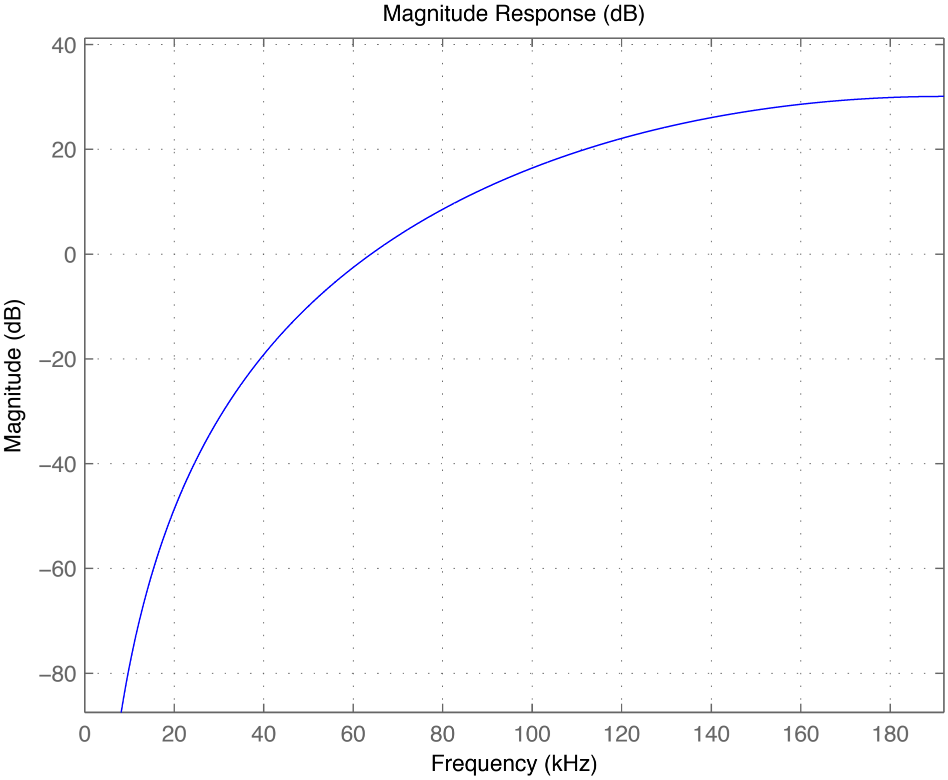


Figure 5.4: Magnitude response of a fifth order filter using calculated coefficients

Once the coefficients have been determined then the system can be simulated using Matlab®'s Simulink tool. Figure 5.5 shows the Simulink configuration for testing the noise shaper.

Putzeys (2006) explained that this method of noise shaping would not be flawless. The output sampling is signal dependent while the input is sampled at consistent intervals. This results in phase-modulating a signal with itself,

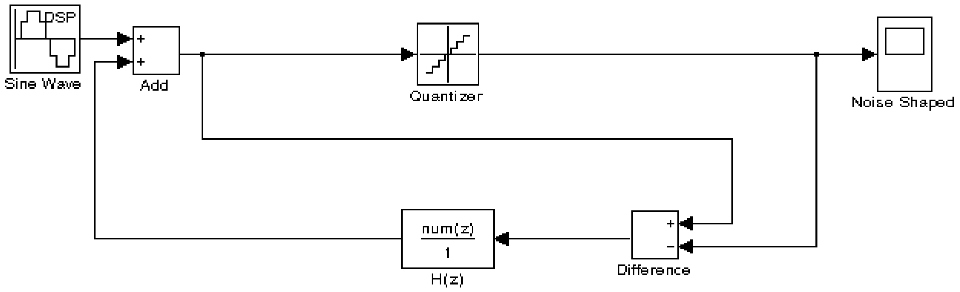


Figure 5.5: Noise shaper Simulink model

which results in distortion in the audio band due to noise folding back into the audio band. High order noise shaping with this configuration can actually increase distortion more than reduce it in the baseband.

5.2.1 Improvement through preprocessing

Goldberg *&* Sandler (1991*b*) presented a pre-modulation technique which could eliminate all audio-band distortion. The technique was dubbed pseudo natural pulse width modulation (PNPWM). The concept is to achieve the performance attained when using natural PWM while still using a sampled UPWM. The low-frequency distortion is corrected for by estimating how a continuous time equivalent would respond in the same situation (Putzeys, 2006)

The cross-point where a naturally sampled carrier waveform would intersect with a reference waveform is calculated. Evenly spaced samples are taken and then an n-th order polynomial is fitted through the points. The comparison is made between the n-th order polynomial and a first order linear equation which represents a sawtooth waveform (Jacobs, 2006). Goldberg *&* Sandler (1991*b*) used the Newton-Raphson method for the approximation.

Stability can become an issue as the demodulation of shaped noise is not corrected for by this scheme (Putzeys, 2006). This results in the design of the noise transfer function to be a very intricate process to avoid instability.

5.3 Digital noise shaping loop filter

The loop filter, sometimes referred to as a compensator, is a useful tool when high performance is required from a low resolution modulator. This is the trademark of a noise shaper and is therefore fitting for this application.

5.3.1 Compensator

The term compensator is often used in control theory (Wescott, 2006:pg. 125). It refers to an element of the control system that corrects an element of the plant's behaviour, in the case of this project the plant would be the pulse width modulator.

Cascade compensation is a very common control system topology. Figure 5.6 shows cascade compensation where the error signal is found and the control signal is created entirely from the error signal. H is the compensator and G is the plant.

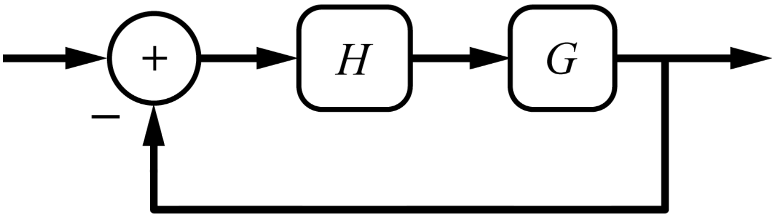


Figure 5.6: Cascade compensation adapted from Wescott (2006)

The system transfer function for this configuration is

$$H_s = \frac{HG}{1 + HG} \; . \tag{5.3.1}$$

Feedforward compensation operates slightly different to cascade compensation. The reference signal is combined through a filter with the control signal without being affected by the feedback. This can increase a system's transient response without affecting its stability. Figure 5.7 shows the feedforward configuration.

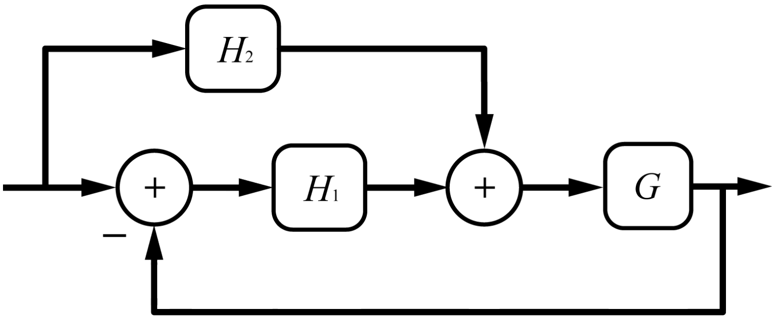


Figure 5.7: Feedforward compensation adapted from Wescott (2006)

The system transfer function for this configuration is

$$H_s = \frac{(H_1 + H_2)G}{1 + H_1 G} \ . \tag{5.3.2}$$

Feedback compensation modifies the plant output before it is compared to the control signal. Figure 5.8 shows this design. This reason why this topology would be implemented is to modify a property of the system which is not actually being measured directly. Wescott (2006) gave an example of measuring the position of a system and then controlling velocity.

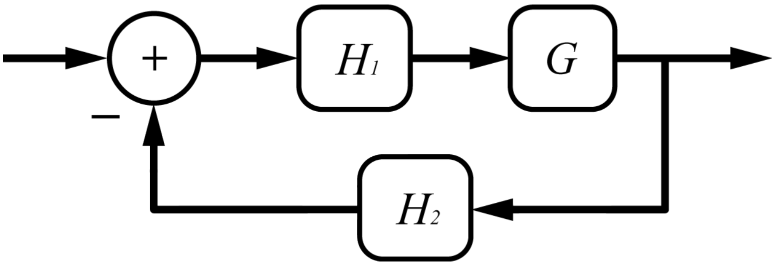


Figure 5.8: Feedback compensation adapted from Wescott (2006)

The transfer function of this system configuration is

$$H_s = \frac{H_1 G}{1 + H_1 H_2 G} \ . \tag{5.3.3}$$

5.3.2 Loop filter

Making use of a digital loop filter helps to combat the problems associated with a classically configured noise shaper. The performance which can be obtained when using a loop filter is agreeable when trying to reproduce high resolution audio.

In order to suppress noise and distortion, the elementary pulse width modulator can be embedded into a feedback loop with a loop filter which provides high loop gain in the audio-band. The high gain in the audio-band results in high error suppression (Risbo, 2005). It also typically has a less than unity gain at the switching frequency (Mouton & Putzeys, 2009).

The implementation of this filter configuration is easily realisable as long as the design of the filter coefficients is effectuated with care. Stability is however much easier to achieve than when designing a classic noise shaper as discussed in Section 5.2.

5.3.3 Loop filter design

The loop filter can be designed using Matlab®'s SISOTOOL component. SISOTOOL stands for single-in single-out tool and is part of the control systems toolbox of Matlab®. It enables the user to design a filter with the understanding that the filter will be used in some kind of a loop, such as a negative feedback loop. The SISOTOOL filter designer verifies whether a particular filter design will be stable or not.

The first part of the design is to achieve the large gain required throughout the audio band. This can be fulfilled by placing a pole at zero Hz (DC). The gain can then be adjusted within SISOTOOL to the desired level. Choosing a value of 100 dB at DC was shown to provide the necessary amount of gain (Mouton & Putzeys, 2009). Shown in Figure 5.9 is the bode plot of a single pole placed at DC.

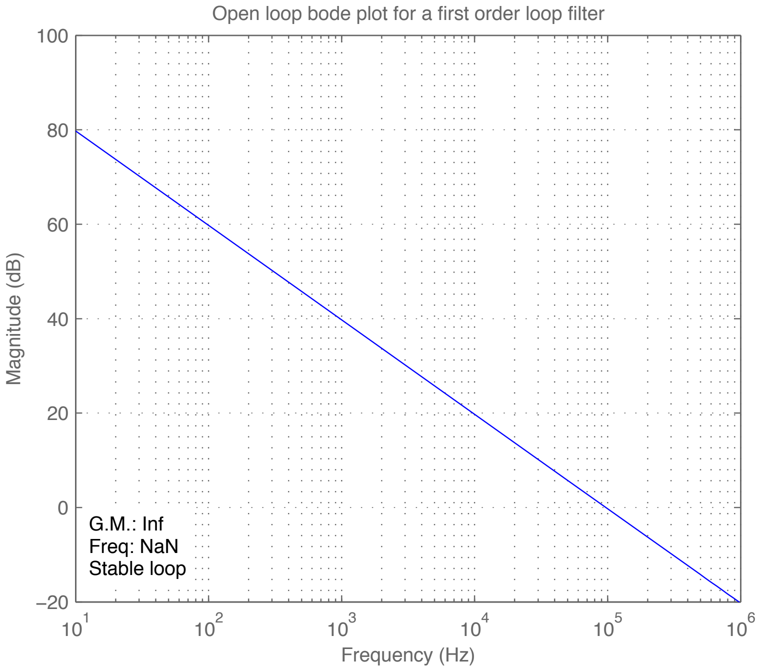


Figure 5.9: Open loop bode plot for a first order filter

The pole causes a downward sloping function which crosses the 0 dB point at 100 kHz. This ensures stability as it is sufficiently lower than the PWM switching frequency which is chosen as 384 kHz. The sampling frequency was chosen as 49.152 MHz which is a multiple of 384 kHz. In order for the function to cross the 0 dB point at the correct frequency, the gain had to be adjusted from 100 dB to 80 dB. This configuration is not ideal as the gain in the audio band drops off very quickly.

Mouton & Putzeys (2009) and Schreier & Temes (2005:pg. 101) showed that the sharp slope which is shown in Figure 5.9 can be flattened out in the audio band by placing a complex pole pair at a higher frequency. An associated zero pair is required to stabilise the system once the pole pair is introduced. Figure 5.10 shows the bode plot of such a system. This is referred to as a third order system due to the single pole at DC and then the complex pole pair which have been placed on the unit circle at 6.3 kHz. The accompanying complex zero pair was placed at 54.818 kHz with a damping ratio of 0.993.

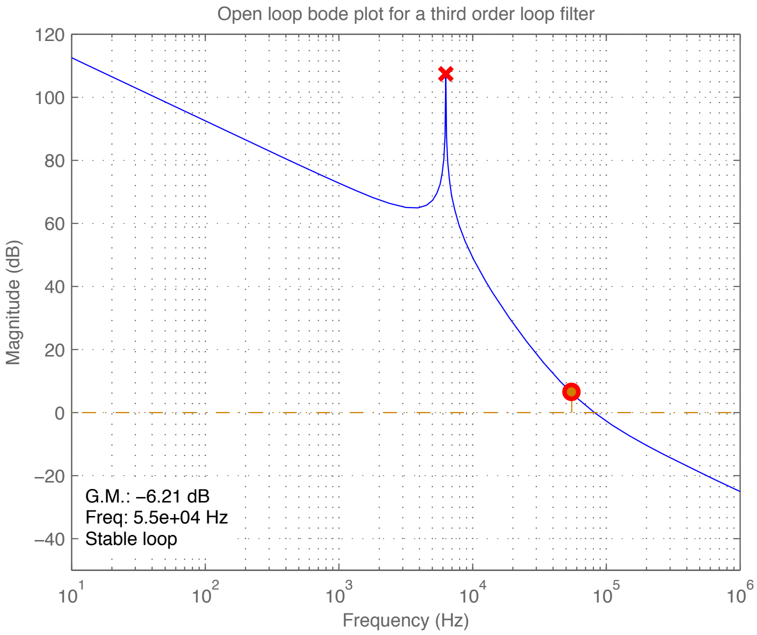


Figure 5.10: Open loop bode plot of a third order filter

The shape of the filter can be further improved by adding another pole pair and their corresponding zeros. This would then make the system a fifth order loop filter. Figure 5.11 shows the resulting bode plot when pole pairs are placed at 7.5 kHz and 16 kHz. Their corresponding zeros were placed at 30 kHz and 40 kHz, both with a damping ratio of 0.707. As can be seen in Figure 5.10 and Figure 5.11 the gain can be further increased as the order of the filter increases, there is however a limit of how much gain can be applied before saturation of the modulator occurs.

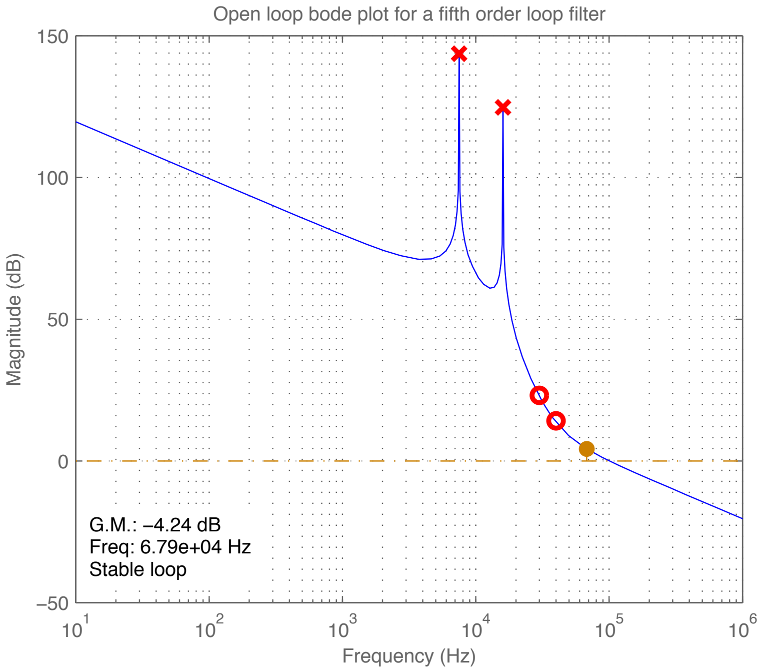


Figure 5.11: Open loop bode plot of a fifth order filter

In order to view what effect the filter has on noise in the system, the Noise Transfer Function (NTF) of the filter can be plotted. In this case it is defined as

$$NTF = \frac{1}{1 + H(z)} \ . \tag{5.3.4}$$

Quantisation noise in the audio band is suppressed due to the high gain in the audio band which was specified in the design of the filter.

Figure 5.12 and Figure 5.13 show bode plots for the NTF of the loop filters previously depicted in Figure 5.10 and Figure 5.11 respectively.

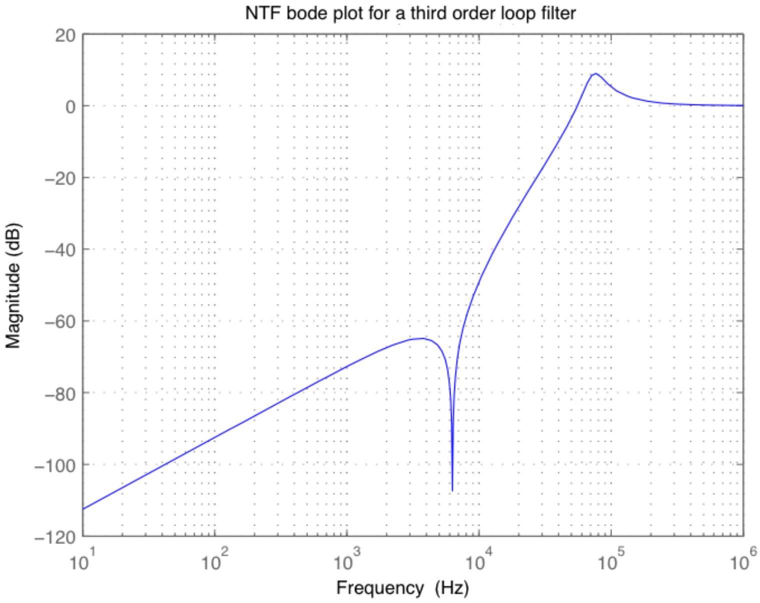


Figure 5.12: NTF bode plot of a third order filter

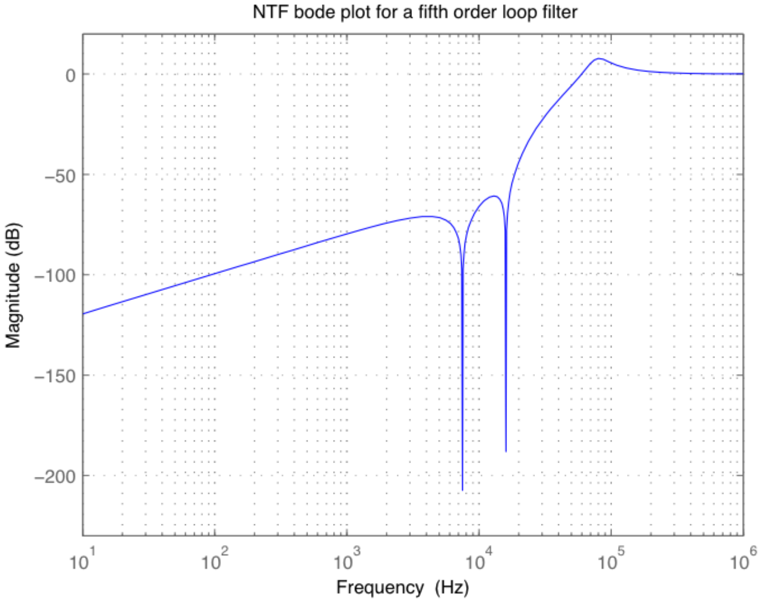


Figure 5.13: NTF bode plot of a fifth order filter

5.3.4 Loop filter verification

Matlab®'s Simulink tool provides a good platform for testing the loop filter. The entire system with pulse width modulation and feedback can be implemented. The evaluation was performed using an upsampled sine wave input at 10 kHz. The sine wave was selected to have a sample rate of 12.288 MHz as the interpolation is assumed to have been performed by the time the input signal reaches the noise shaping section. The amplitude of the sine wave was chosen as 0.9 of full scale.

In order for a comparison to be made, the results of using a pulse width modulator without feedback and a loop filter need to be known. Figure 5.14 shows the Simulink model for a basic pulse width modulator.

The sawtooth carrier is generated by using a 7-bit counter which is scaled so that its output goes from -1 to $+1$. The output of the pulse width modulator is also scaled to give a -1 and $+1$ output amplitude, which is relevant when feedback is employed. The scaling is important to keep all the signals within the same operating boundaries. Figure 5.15 shows the spectral content of the resulting PWM waveform from Figure 5.14. The switching frequency can be clearly seen at 384 kHz, along with its sideband harmonics spaced at 10 kHz apart, due to the fundamental 10 kHz input signal. Figure 5.16 shows a zoomed view of Figure 5.15, focus is towards the audio band.

The first order loop filter can now be added with negative feedback to evaluate the system's performance. Figure 5.17 shows the Simulink model for the pulse width modulator with a first order loop filter. The spectral content of the resulting PWM waveform is shown in Figure 5.18. The zoomed view of Figure 5.18 is shown in Figure 5.19. There is still a large amount of harmonics in the audio band due to the filter's sharp roll-off which does not provide enough gain throughout the audio band.

When testing the higher order filters, the same Simulink configuration can be used as with the first order filter. Figures 5.20 and 5.21 show the spectral contents of the PWM output when a third and fifth order loop filter is used, respectively while Figures 5.22 and 5.23 show their zoomed versions.

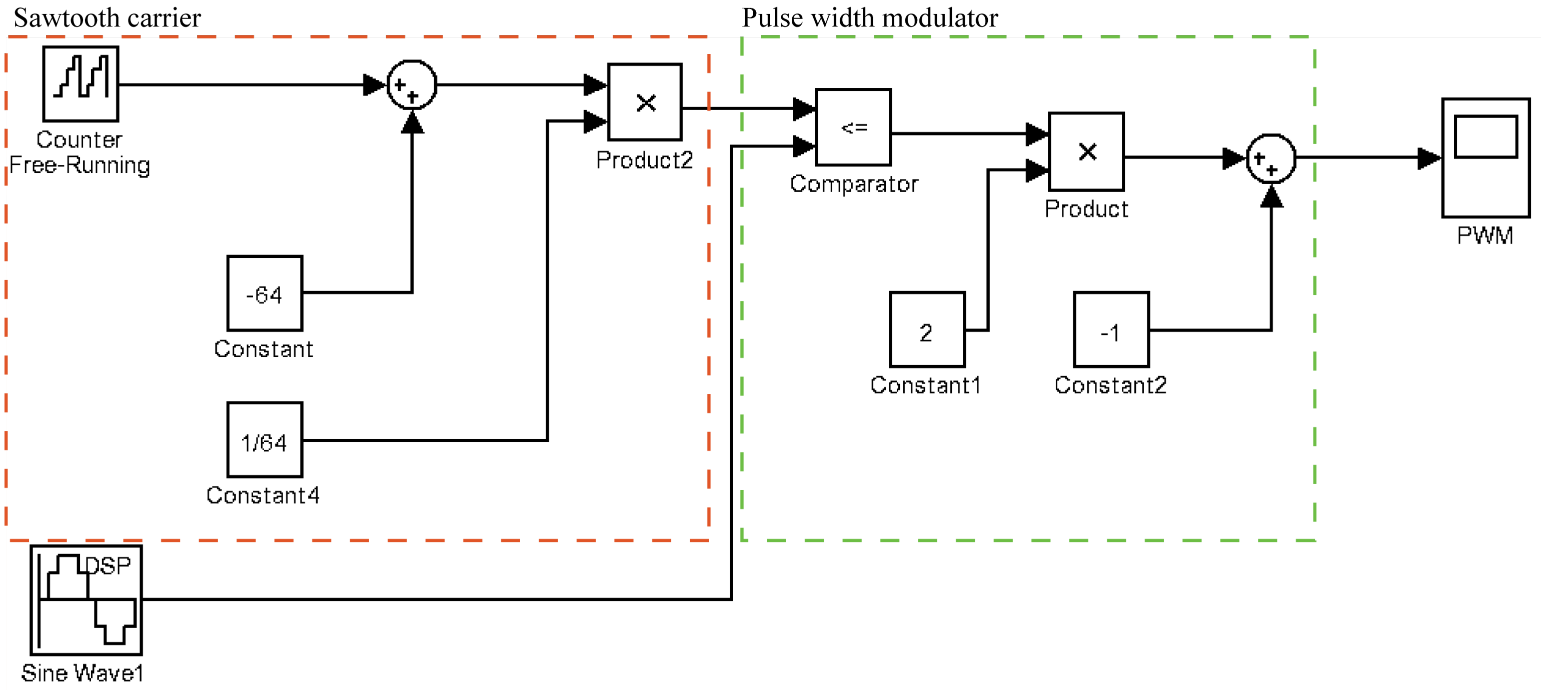


Figure 5.14: Simulink model for a pulse width modulator

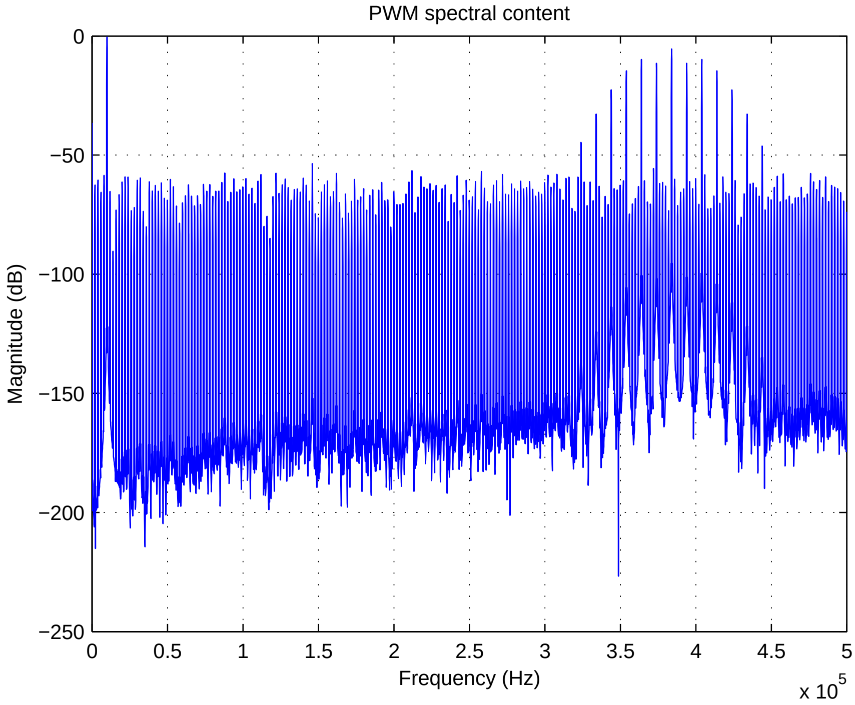


Figure 5.15: Spectral content of a 7-bit PWM waveform

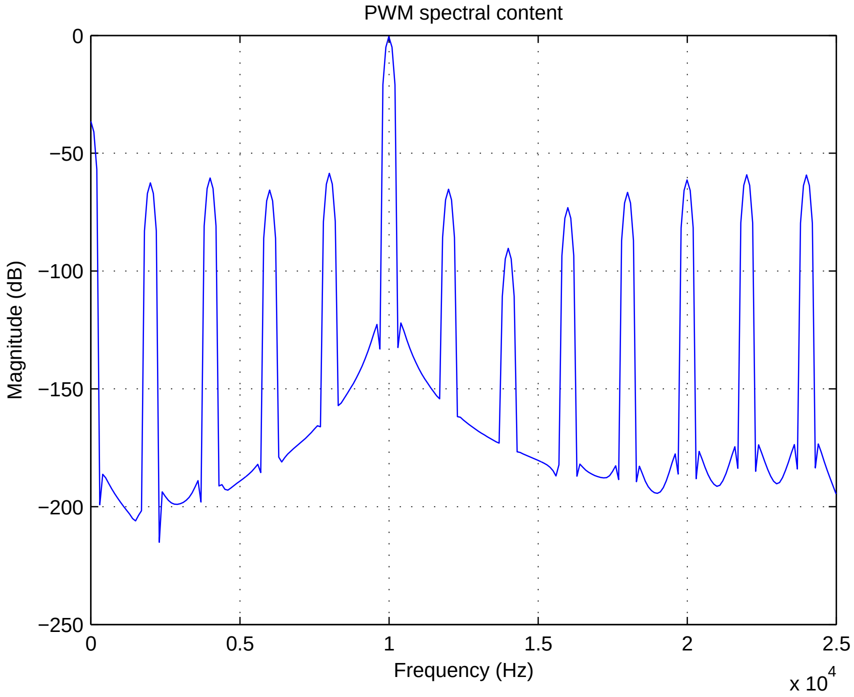


Figure 5.16: Zoomed view of the audio band of Figure 5.15

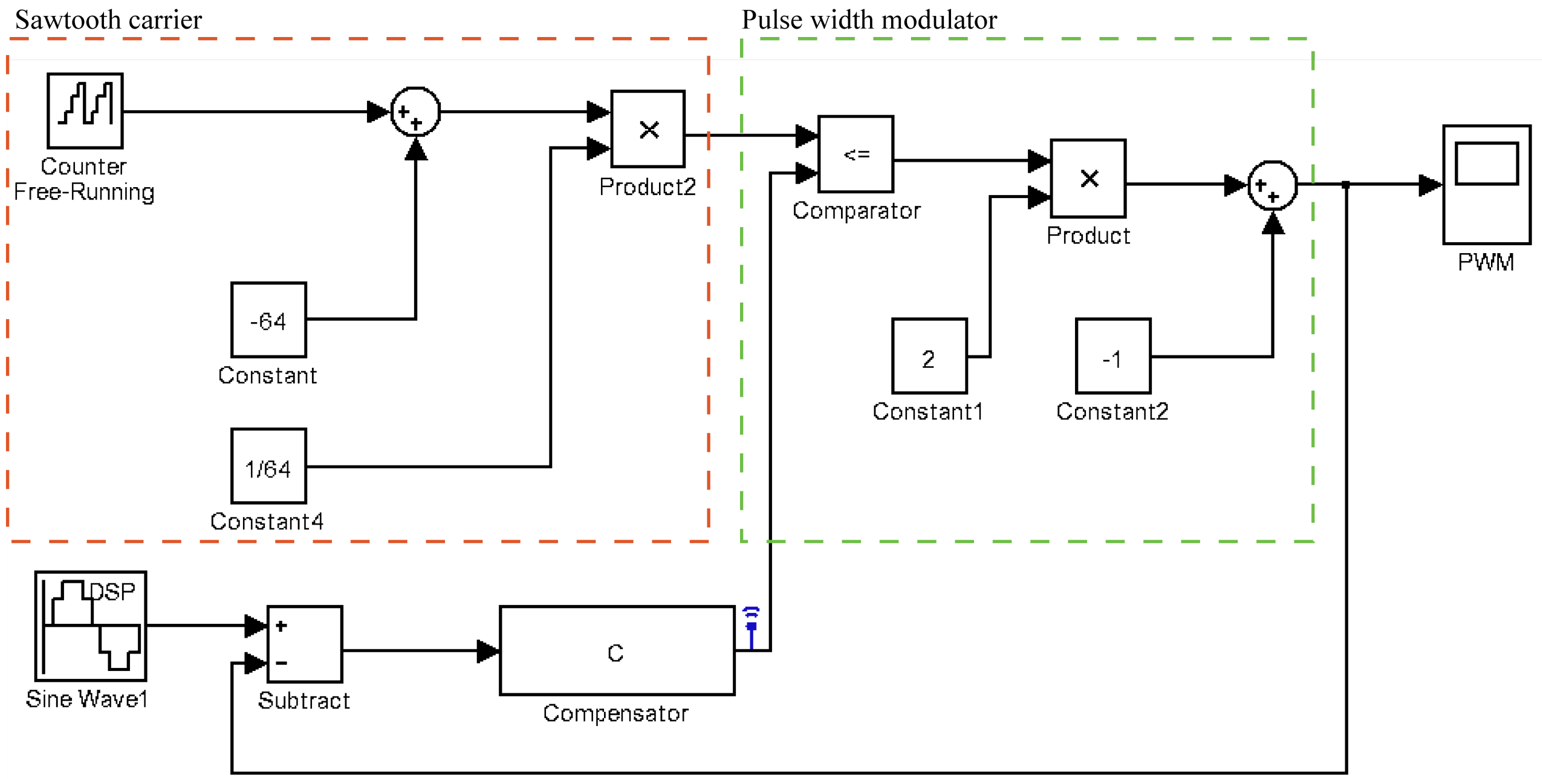


Figure 5.17: Simulink model for a pulse width modulator and loop filter

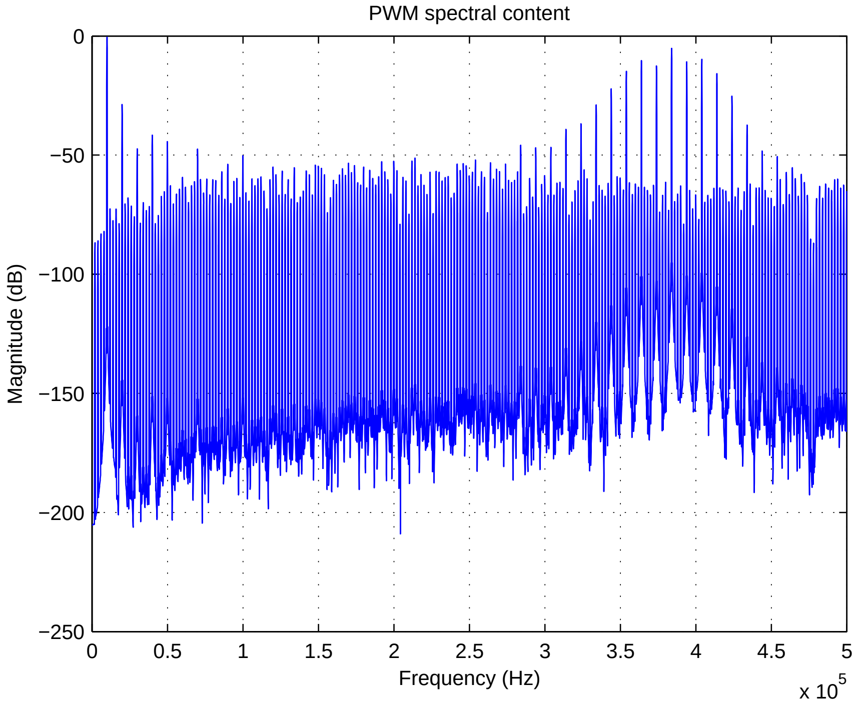


Figure 5.18: Spectral content of a 7-bit PWM waveform with a first order filter

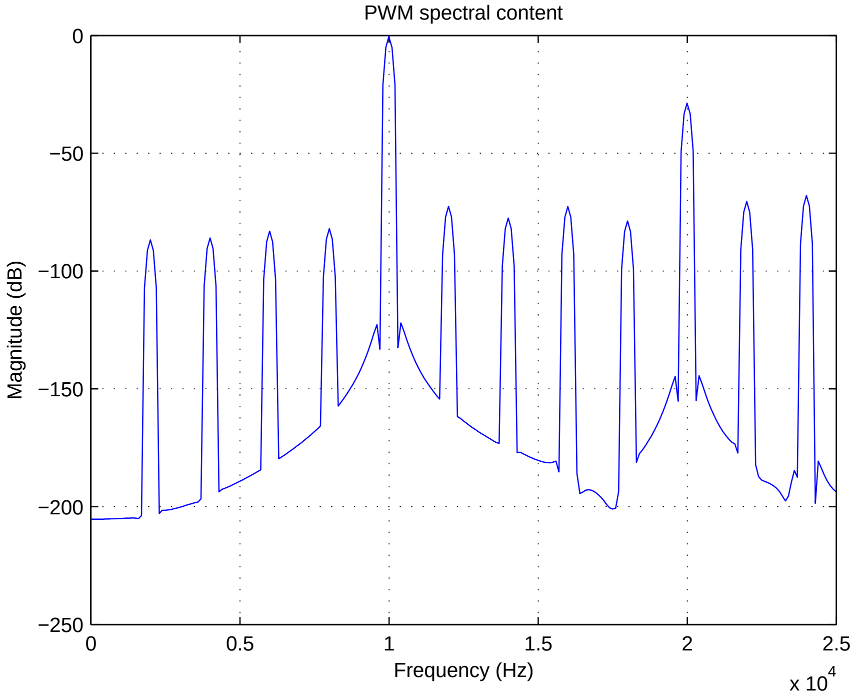


Figure 5.19: Zoomed view of the audio band of Figure 5.18

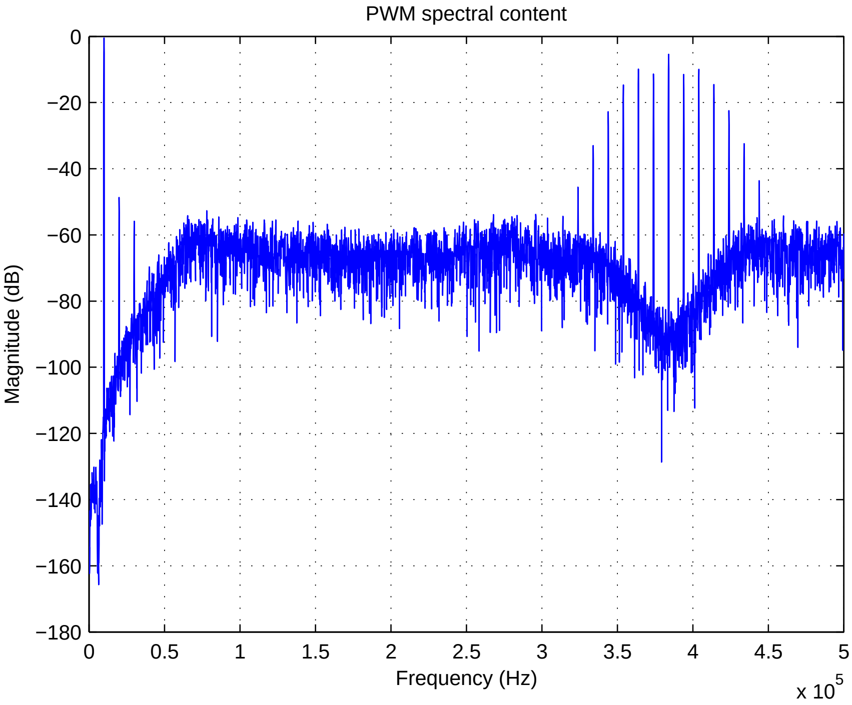


Figure 5.20: Spectral content of a 7-bit PWM waveform with a third order loop filter

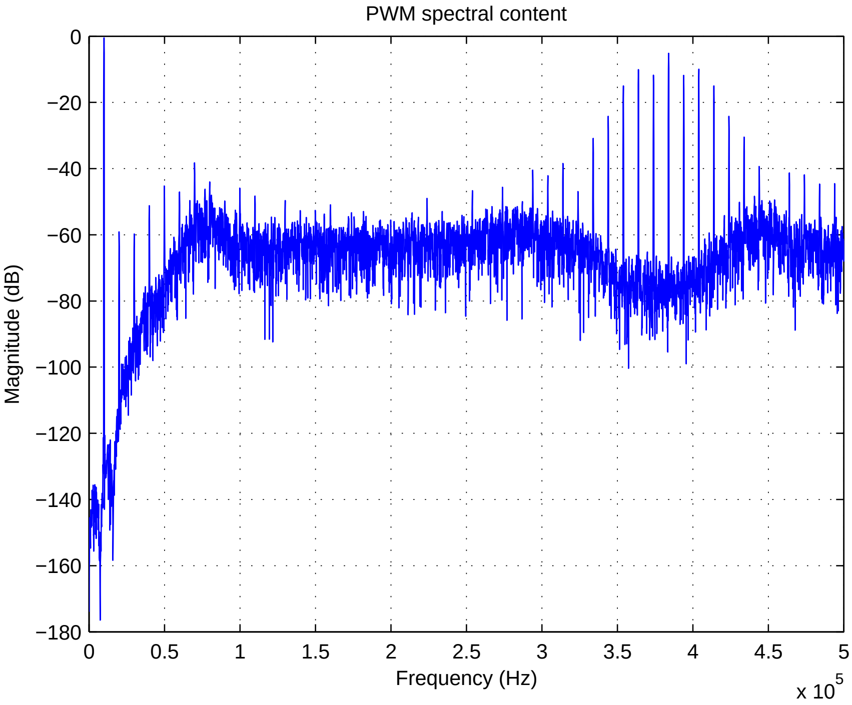


Figure 5.21: Spectral content of a 7-bit PWM waveform with a fifth order loop filter

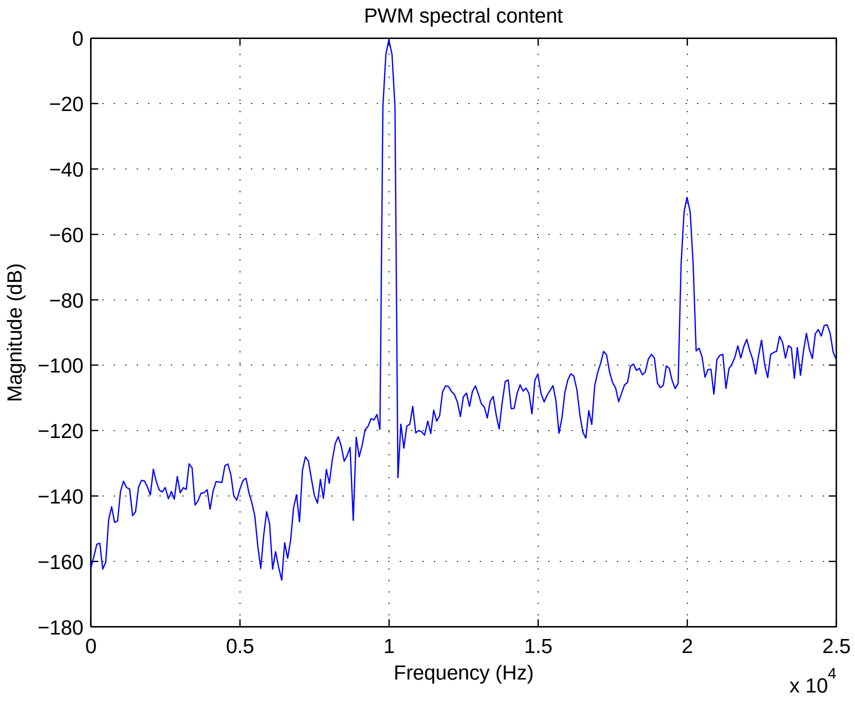


Figure 5.22: Zoomed view of the audio band of Figure 5.20

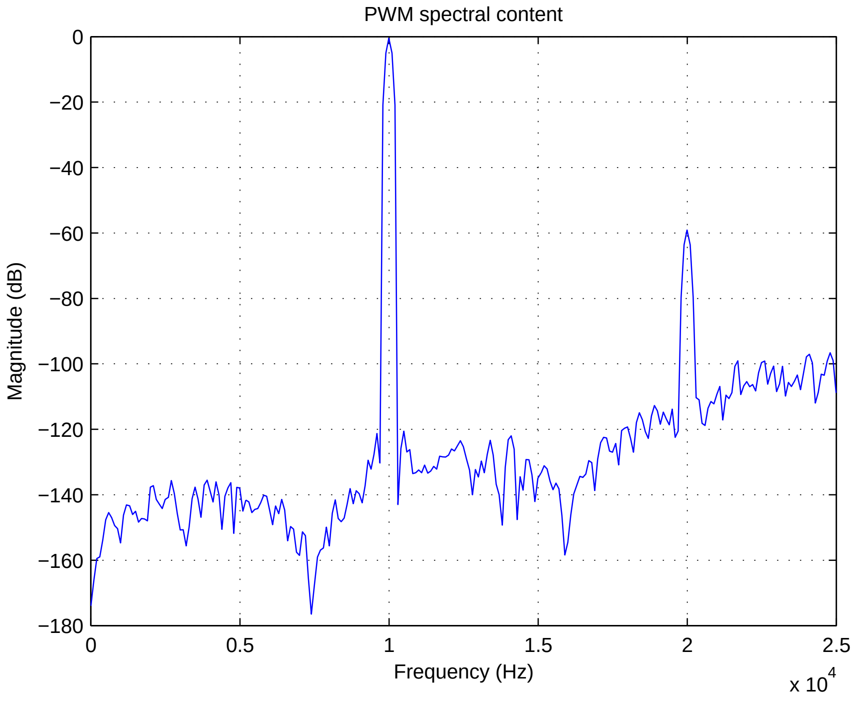


Figure 5.23: Zoomed view of the audio band of Figure 5.21

When viewing the spectral content in the audio band, it is clear that increasing the order of the loop filter decreases the noise floor. The trend of the noise floor dropping as the filter order is increased cannot continue indefinitely. The noise rejection did not improve significantly more when a seventh order filter was utilised. Employing a higher order filter would not be as beneficial due to the amount of system resources required to do so.

The use of a loop filter is however not perfect as there is noise being folded back into the audio band. This can be observed in Figure 5.22 and Figure 5.23 where there is an anomalous spike at 20 kHz. It had been previously thought that such results are to be expected as the result in a real life situation is never perfect and numerical errors in simulations can be the cause (Neesgaard & Risbo, 2006). There is however distortion present and it can be attributed to the high frequency components of the PWM signal being fed back to the comparator of the pulse width modulator, this is the ripple signal.

This undesirable situation can be remedied by including a ripple compensation scheme. Risbo (2005) explained that ripple instability occurs when a race-around condition becomes active in the system. The comparator of the pulse width modulator prematurely reverts to its previous state due to the ripple feedback pulling the input of the comparator past the zero point. The feedback causes a non linearity of the PWM operation due to the high frequency carrier components being aliased (Mouton & Putzeys, 2009). This eventually leads to distortion of the output signal. The system uses negative feedback to reduce the distortion, yet intrinsically produces its own distortion in the process (Candy & Cox, 2004).

It could be thought that the quick solution would be to increase the loop gain to counteract the aliasing distortion. This would however be void as the increased closed-loop error suppression would be cancelled in the audio band due to the proportionally scaled open-loop distortion (Neesgaard & Risbo, 2006).

The next solution could be to add additional high frequency poles or zeros to the loop filter design. This would commonly just result in more distortion or an unstable system (Neesgaard & Risbo, 2006).

5.4 Ripple compensation

Neesgaard & Risbo (2006) presented a thorough investigation of the distortion which is caused by ripple feedback. There were two parts of the distortion which were identified. The first is that the PWM signal gets phase modulated by a nonlinear function of the signal itself. The harmonic distortion caused by phase modulation will be caused by odd harmonics only, due to the time delay being an even function multiplied by the signal itself, which results in odd harmonics (Neesgaard & Risbo, 2006). The second mechanism consisted of distortion of the pulse width which results in a DC non linearity (Mouton & Putzeys, 2009).

Candy & Cox (2004), Hawksford (2005) , Neesgaard & Risbo (2006) and Putzeys (2006) have presented methods for counteracting the effect of ripple feedback.

Candy & Cox (2004) presented a solution where the carrier is modulated by a small amplitude signal which is proportional to the derivative of the input signal. The period of the carrier wave is increased by the modulating signal. This is by an amount which is proportional to the square of the input signal. This approach effectively eliminates the third harmonic intrinsic distortion (Candy & Cox, 2004).

Hawksford (2005) introduced the nodal transition filter. When the method is viewed in the time domain, stability can be achieved by introducing a set of constant voltage crossover filters. When implemented correctly it can change the node, where feedback is acquired, in a frequency selective manner (Hawksford, 2005). The concept is that the loop filter would receive the same low frequency components that it would have before using a nodal transition filter, yet the high frequency ripple is attenuated by a low pass filter which forms part of the configuration. Neesgaard & Risbo (2006) showed that this approach is not very effective in practice and better results could be achieved by just using a first order integrator modulator instead of the complicated filter configuration.

Neesgaard & Risbo (2006) introduced the Minimum Aliasing Error (MAE) loop filter. Through analysis, it was found that a first order loop filter behaved in a desirable manner, with regards to distortion, yet its performance was less than adequate. The concept of an MAE filter is to reduce the real part of the loop transfer function above the switching frequency. The reasoning behind this is that the real part manages the DC error.

The MAE loop filter was developed by looking at low order filters. An asymptotic cancellation effect was achieved by summing a low-pass filter with a second order integrator. This results in a first order high frequency roll off of the amplitude characteristic which ensures closed-loop stability. The high frequency amplitude is that of a first order, yet the real part has a fourth order shape for the MAE filter. Neesgaard & Risbo (2006) also explained that the dominant contributor of the DC error are the frequency components from the switching frequency fundamental. The open-loop THD, which is the DC error, of the MAE filter is reduced by a factor of

$$L \cong \left(\frac{2\pi f_{sw}}{p}\right)^2 \tag{5.4.1}$$

where p is the pole frequency.

The pole frequency would typically be chosen as $3 - 10$ times below the switching frequency for closed-loop stability. This would result in a reduction of the open-loop distortion of a factor of $10 - 100$ while still achieving the same low frequency loop gain (Neesgaard & Risbo, 2006).

The method offered by Putzeys (2006) is a very simple approach which cancels the unmodulated edge of a single sided modulator. Implementation is much easier than that of the techniques presented by Candy & Cox (2004) or Neesgaard & Risbo (2006). Being able to ignore the unmodulated edge in a single sided modulator is the solution to keeping the ripple constant and in phase.

Figure 5.24 shows the implementation of the scheme presented by Putzeys (2006). The resulting waveforms are shown in Figure 5.25. The method was explained fully by Mouton & Putzeys (2009). The feedback signal is comprised of the modulator output $p(t)$ which is added to the sawtooth carrier $s(t)$. This cancels the unmodulated edge of p(t). The signal which results is a pseudo sawtooth waveform $y(t)$. The time average of y(t) is equal to that of the modulator output $p(t)$. The advantage of this scheme is that the shape of the ripple component of $y(t)$ is independent of the amplitude of the mean modulator input signal $x(t)$. The pseudo sawtooth waveform $y(t)$ is then subtracted from the input i(t) and then passed through the loop filter $G(s)$. The control loop tracks the pulse width modulator input signal $x(t)$. The comparison of $x(t)$ and the carrier $s(t)$ coincide with those of $i(t)$ and $s(t)$ (Mouton & Putzeys, 2009).

What is important to note is that the shape of the ripple component $x(t)$ is independent of the average value of $x(t)$. This reduces the nonlinearity resulting from the ripple component of the PWM input signal and the pulse width modulation process interacting (Mouton & Putzeys, 2009).

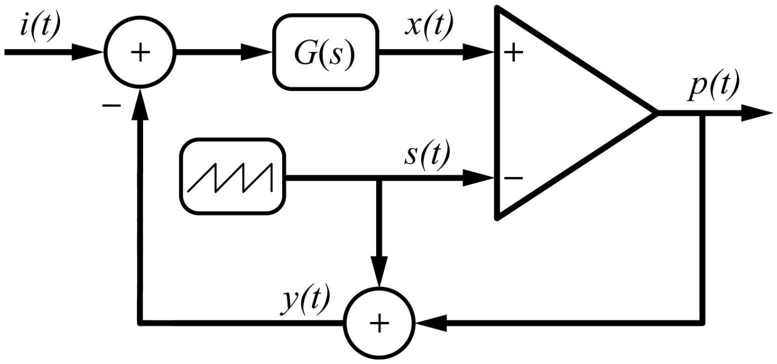


Figure 5.24: Implementation of ripple compensation adapted from Mouton & Putzeys (2009)

5.5 Ripple compensation verification

The ripple compensation scheme presented by Putzeys (2006) was chosen for implementation due to its simplicity. Figure 5.26 shows the Simulink model for the pulse width modulator, loop filter and ripple compensator.

Analysing the spectral content of the PWM signal is the true verification of whether the ripple compensation is an effective treatment for the unwanted distortion in the audio band. Figure 5.27 shows the spectral content of the output PWM signal of Figure 5.26 when a third order loop filter is used. Figure 5.28 shows a zoomed view of Figure 5.27, with an emphasis on the audio band. Figure 5.29 shows the spectral content when a fifth order filter is used and Figure 5.30 shows a zoomed view of Figure 5.29.

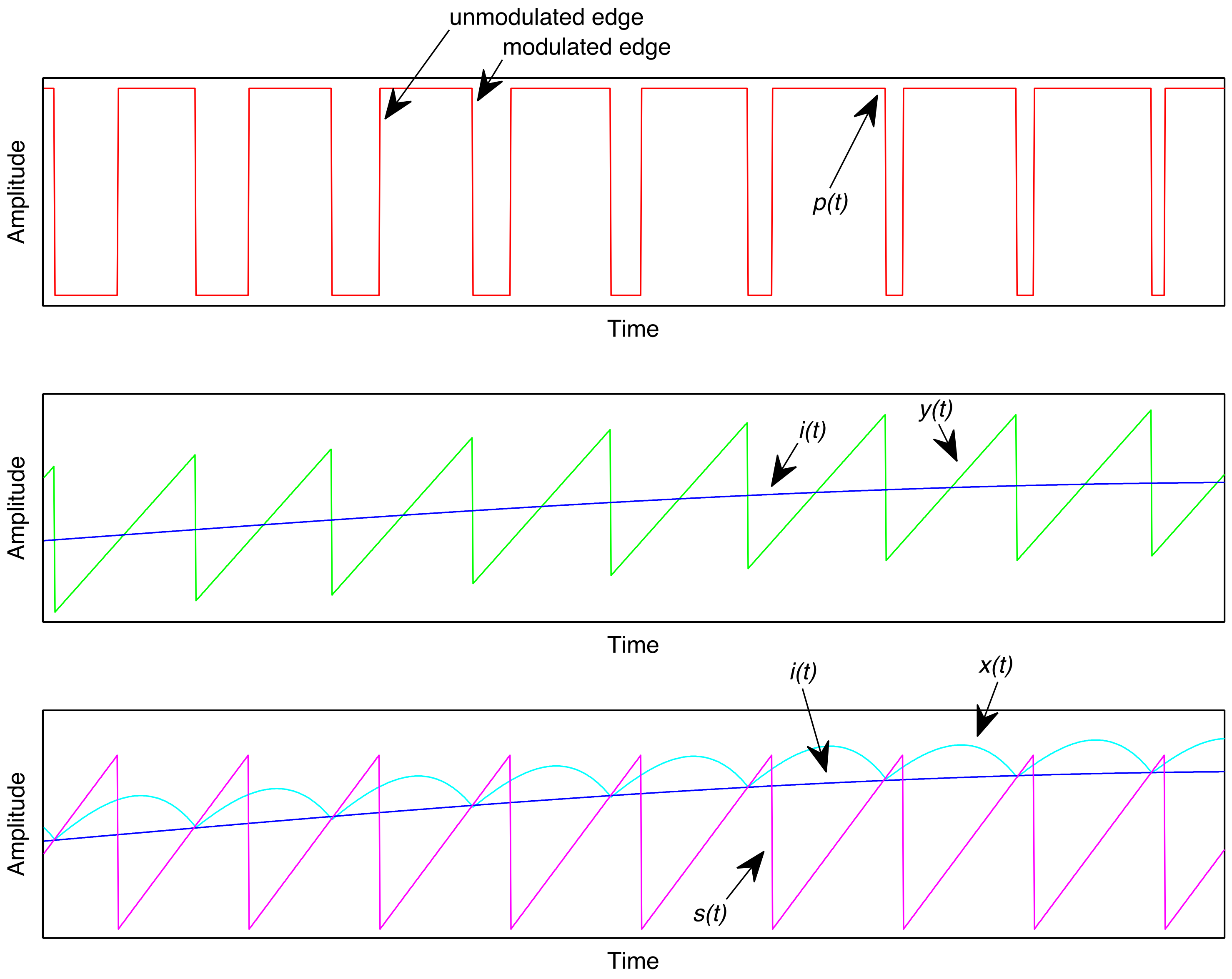


Figure 5.25: Ripple compensation waveforms adapted from Mouton & Putzeys (2009)

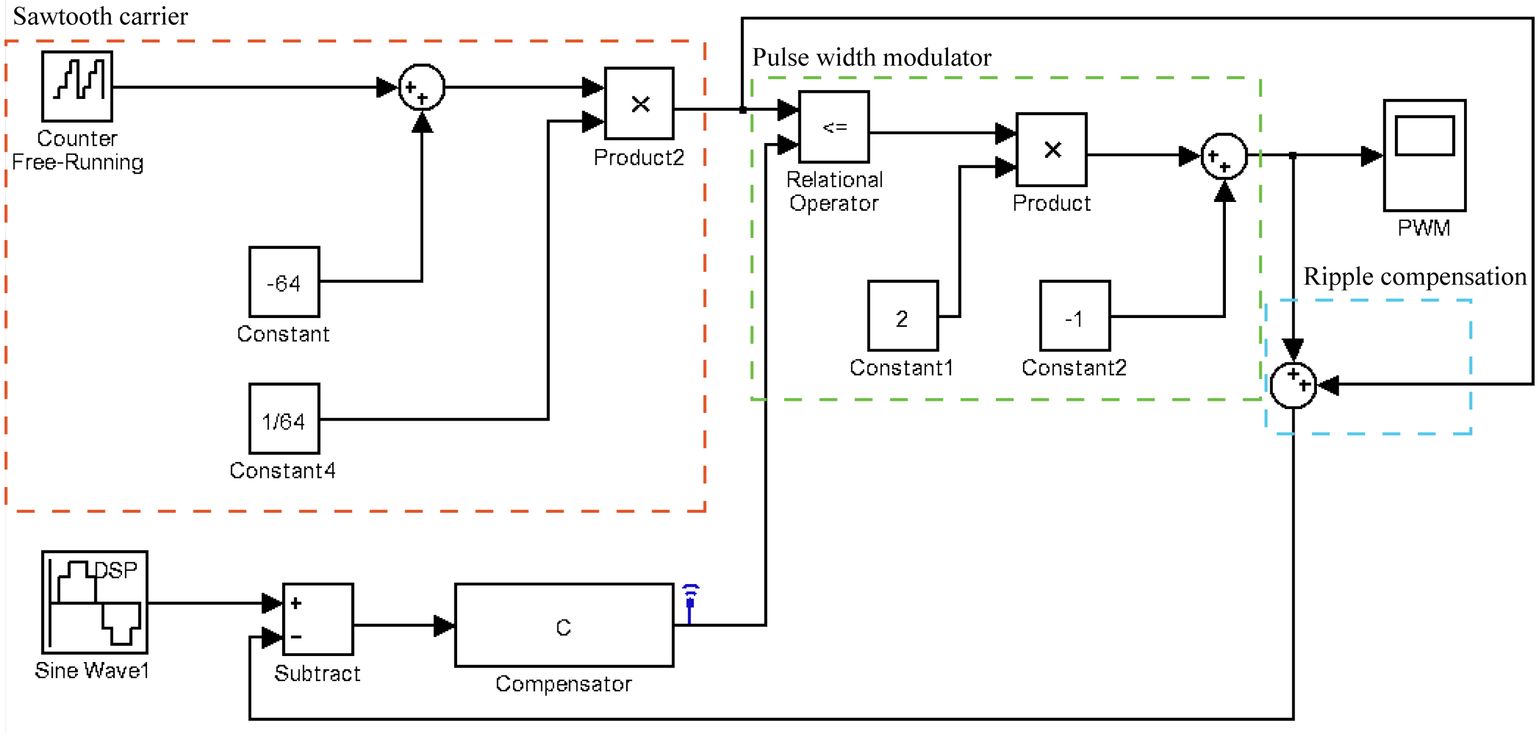


Figure 5.26: Simulink model for a pulse width modulator and loop filter including ripple compensation

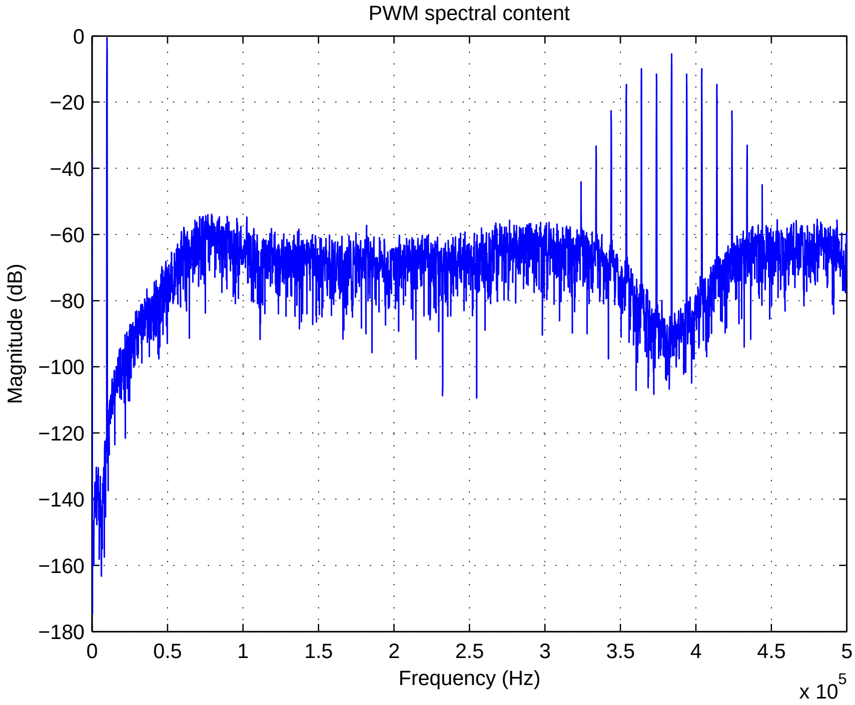


Figure 5.27: Spectral content of a 7-bit PWM waveform with a third order loop filter and ripple compensation

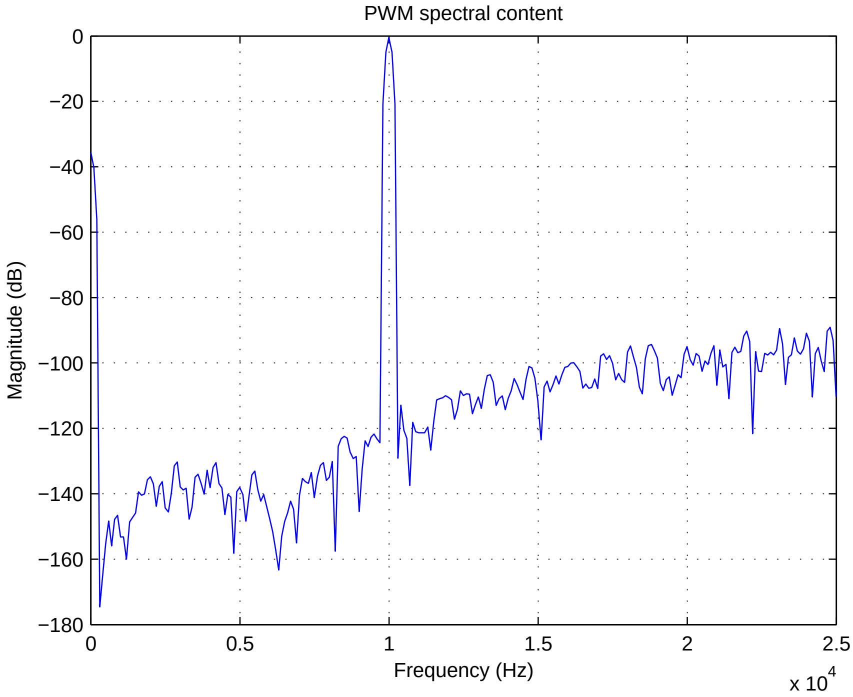


Figure 5.28: Zoomed view of the audio band of Figure 5.27

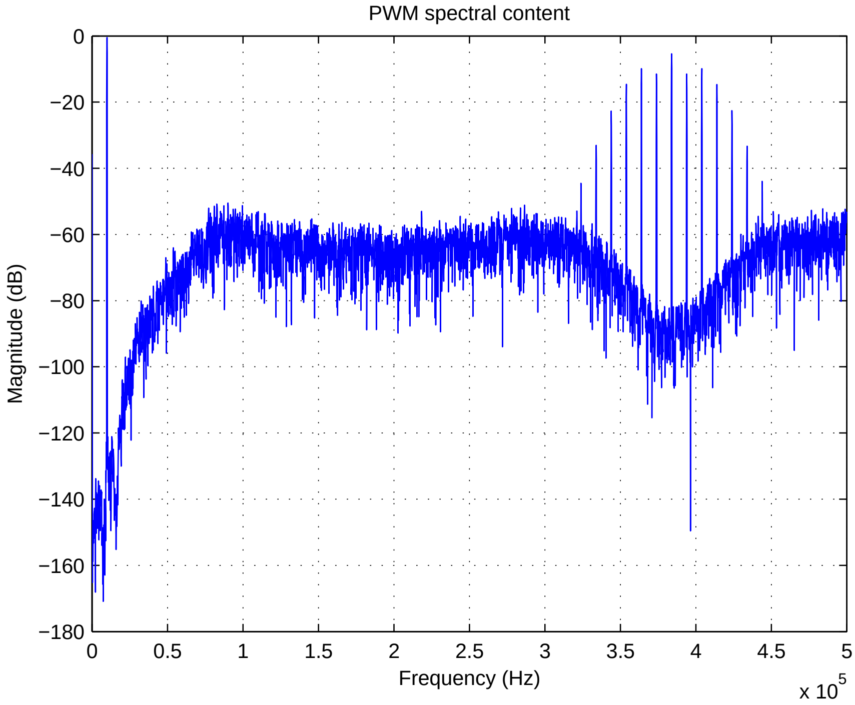


Figure 5.29: Spectral content of a 7-bit PWM waveform with a fifth order loop filter and ripple compensation

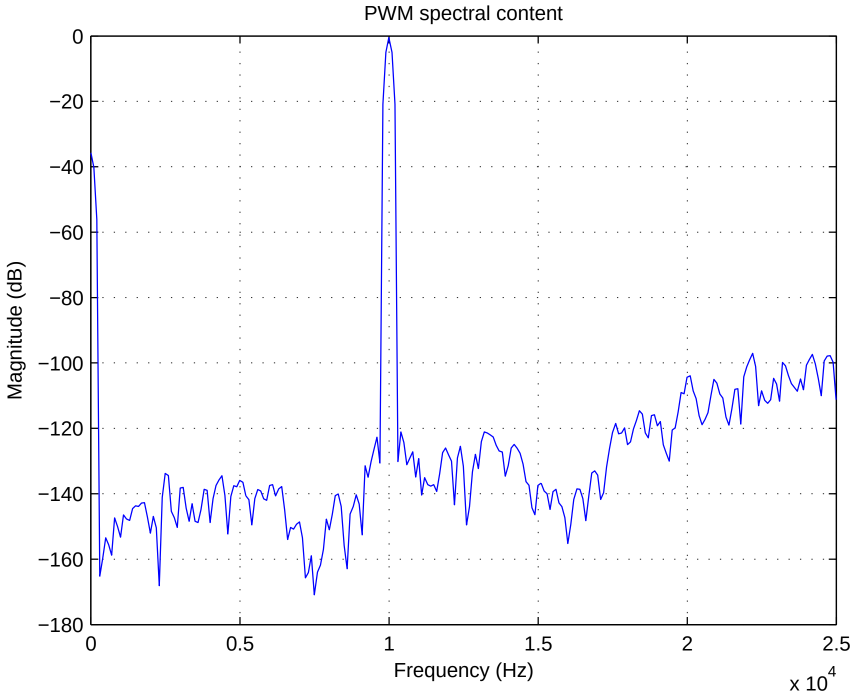


Figure 5.30: Zoomed view of the audio band of Figure 5.29

5.6 Loop filter implementation

The transfer functions of the various order loop filters can be exported from SISOTOOL and viewed in Matlab®. The transfer function would have to be implemented in a filter structure for it to be realisable. When exported from SISOTOOL the transfer function is of the form

$$H(z) = \frac{b_m z^m + b_{m-1} z^{m-1} + \ ... \ + b_1 z + b_0}{a_n z^n + a_{n-1} z^{n-1} + \ ... \ + a_1 z + a_0} \tag{5.6.1}$$

from which specific filter coefficients need to be determined, depending on the filter structure.

Many loop filter configurations have been presented by Norsworthy *et al.* (1997), Kozak & Kale (2003), Schreier & Temes (2005) and Maloberti (2007). The various structures which they present have been used very successfully when implementing delta-sigma modulators which use loop filters as the heart of their operation.

The transfer function for the first order loop filter when exported from SISOTOOL is

$$H(z) = \frac{0.012784}{z-1} \ . \tag{5.6.2}$$

Equation 5.6.2 is simply an integrator with a scaling factor of 0.012784. An integrator can be configured either as shown in Figure 5.31 or as shown in Figure 5.32. The difference between the two configurations is a delay being placed either in the feedforward or feedback path. Figure 5.31 will feed a delayed version of its output onwards, while Figure 5.32 will have a non-delayed version available.

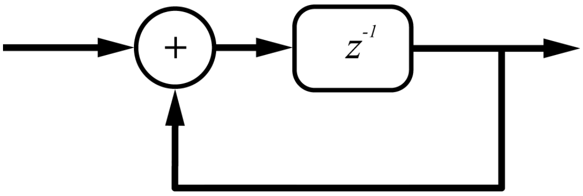


Figure 5.31: Delayed output integrator

The transfer function for the integrator shown in Figure 5.31 is

$$H(z) = \frac{1}{z-1} = \frac{z^{-1}}{1-z^{-1}} \ . \tag{5.6.3}$$

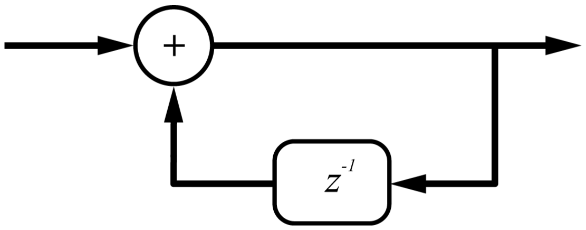


Figure 5.32: Non-delayed output integrator

The transfer function for the integrator shown in Figure 5.32 is

$$H(z) = \frac{z}{z-1} = \frac{1}{1-z^{-1}} \ . \tag{5.6.4}$$

Schreier & Temes (2005) listed the most commonly used loop filter structures as being:

- Cascade of integrators with feedback summation (CIFB),
- cascade of resonators with feedback summation (CRFB),
- cascade of integrators with feedforward summation (CIFF) and
- cascade of resonators with feedforward summation (CRFF).

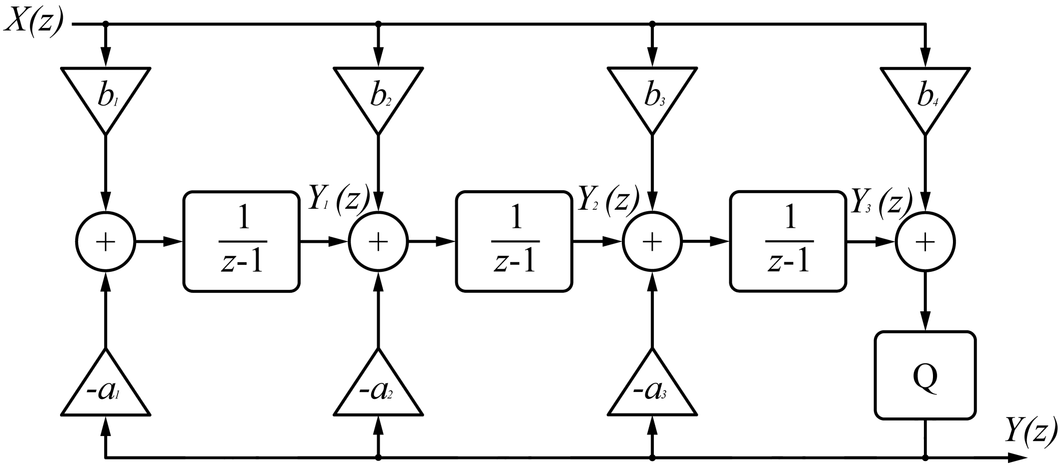


Figure 5.33: Cascade of integrators with feedback summation adapted from Schreier & Temes (2005)

The CIFB structure contains a chain of N delaying integrators. The feedback and input signal are fed to each integrator input port with different scaling factors a_i and b_i. This structure is shown in Figure 5.33 for when $N = 3$.

The Noise Transfer Function (NTF) of the CIFB configuration is of the form

$$H(z) = \frac{(z-1)^N}{a_1 + a_2(z-1) + \ \dots\ + a_N(z-1)^{N-1} + (z-1)^N} \ . \tag{5.6.5}$$

Thus all the zeros of the NTF for this structure must lie at $z = 1$, which is DC. The scaling factors a_i are used to introduce finite nonzero poles into the NTF. The Signal Transfer Function (STF) is

$$G(z) = \frac{b_1 + b_2(z-1) + \ \dots\ + b_{N+1}(z-1)^N}{a_1 + a_2(z-1) + \ \dots\ + a_N(z-1)^{N-1} + (z-1)^N} \ . \tag{5.6.6}$$

This denotes that the zeros of the STF are determined by b_i and its poles by a_i. If a STF of 1 is required, then $b_i = a_i$ can be chosen for all $i \leq N$ and $b_{N+1} = 1$.

As indicated by Equation 5.6.5, the CIFB structure can have the NTF zeros only at DC ($z = 1$). It was shown in Section 5.3.4 that performance is improved if these zeros are located at nonzero frequencies on the unit circle. By modifying the CIFB structure to include a resonator, it is then capable of realising NTF zeros as complex conjugate pairs on the unit circle.

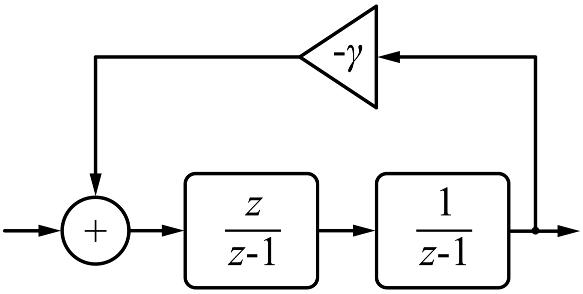


Figure 5.34: Resonator for a loop filter adapted from Norsworthy *et al.* (1997)

Norsworthy *et al.* (1997) explained that adding a small amount of negative-feedback around the integrator pairs of the loop filter can be very beneficial. This resonator configuration is shown in Figure 5.34. In order to ensure that the poles stay on the unit circle, one of the integrators in each resonator needs to be delay free (Schreier *&* Temes, 2005:pg. 119). It then becomes possible to move the open-loop poles up from DC. The open-loop poles become the NTF zeros when the loop is closed. This causes the frequencies where there is high loop gain, which becomes high noise attenuation, to be shifted away from DC to positive frequencies. The transfer function for a pair of integrators with feedback is

$$\begin{aligned} R(z) =& \frac{\frac{z}{z-1} \cdot \frac{1}{z-1}}{1 + \gamma \cdot \frac{z}{z-1} \cdot \frac{1}{z-1}} \\ =& \frac{z}{(z-1)^2 + \gamma z)} \qquad (5.6.7) \\ =& \frac{z}{z^2 + (-2+\gamma)z + 1} \ . \end{aligned}$$

The CRFB structure is shown in Figure 5.35 where the second and third integrators, combined with the feedback path $-\gamma_1$ form the resonator. It contains a complex pole which is the zero of $z^2 + (-2+\gamma_1)z + 1$. It is conventional to have a plain integrator as the input stage in order to minimise the input-referred contribution of noise sources from ensuing stages (Schreier & Temes, 2005). The resonators are inherently unstable, as inferred by the fact that their poles are located on the unit circle. Local oscillations are prevented by placing them into a stable feedback system.

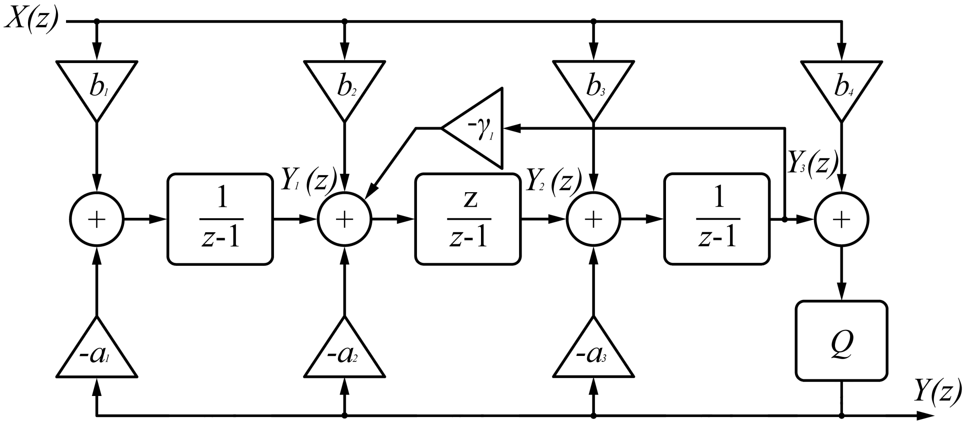


Figure 5.35: Cascaded resonator with feedback summation adapted from Schreier & Temes (2005)

The signal paths which are required to create the zeros of the NTF can also be realised by using feedforward rather than feedback. Shown in Figure 5.36

is a configuration of cascaded integrators with feedforward branches, CIFF. As was the functioning of the CIFB structure, the CIFF's NTF contains its zeros at DC (Schreier *&* Temes, 2005). This leads to the implementation of resonators if optimised zeros are required.

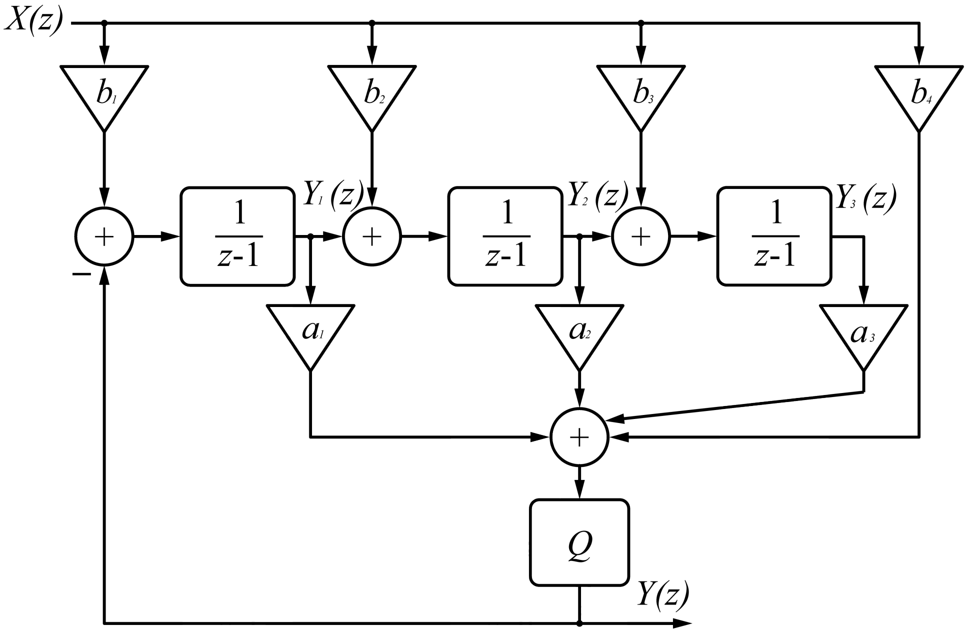


Figure 5.36: Cascaded integrators with feedforward summation adapted from Schreier *&* Temes (2005)

Figure 5.37 shows a chain of integrators with feedforward summation and local resonator feedbacks. What is important to note is that there are no b_i scaling coefficients. When implementing this structure in a real-world situation, the complexity and amount of multipliers is reduced.

In order to determine the coefficients a_i and γ_i for the CRFF filter structure, the transfer function of the filter needs to be compared to that of the filter designed in Matlab®.

The transfer function for the individual sections of the CRFF third order loop filter shown in Figure 5.37 is

$$\frac{Y(z)}{X(z)} = \frac{a_1}{z-1} + \frac{a_2 z}{z^2(-2+\gamma_1)z+1} + \frac{1}{z-1} \cdot \frac{a_3 z}{z^2(-2+\gamma_1)z+1} \tag{5.6.8}$$

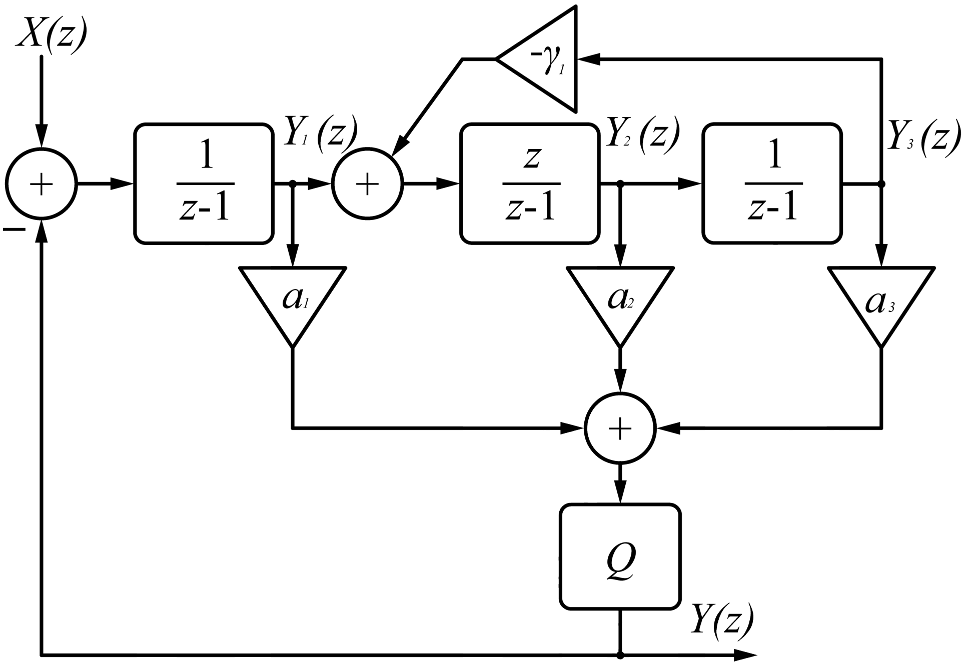


Figure 5.37: Cascaded resonator with feedforward summation adapted from Norsworthy *et al.* (1997)

When the individual sections are combined, they form the transfer function

$$\frac{Y(z)}{X(z)} = \frac{a_1z^2 - 2a_1z + a_1\gamma_1z + a_2z^2 - a_2z + a_3z + a_1}{z^3 - 3z^2 + \gamma_1z^2 + 3z - \gamma_1z - 1} \ . \tag{5.6.9}$$

In order to make the comparison between the Matlab® generated transfer function and that of the filter much easier, the filter structure is combined in terms of z which yields

$$\frac{(a_1 + a_2)z^2 + (-2a_1 + a_1\gamma_1 - a_2 + a_3)z + a_1}{z^3 + (-3 + \gamma_1)z^2 + (3 - \gamma_1)z - 1} \ . \tag{5.6.10}$$

The loop filter transfer function which was exported from Matlab® is

$$\frac{b(z)}{a(z)} = \frac{0.0074761z^2 - 0.014844z + 0.00736884}{z^3 - 2.9999958z^2 + 2.9999958z - 1} \tag{5.6.11}$$

If Equation 5.6.10 and Equation 5.6.11 are equated the result yields the filter coefficients

$$\begin{aligned}\mathbf{a_1} =& 0.007368840280018\\ \mathbf{a_2} =& 0.000107296127841998\\ \mathbf{a_3} =& 3.569351756209371 \times 10^{-7}\\ \gamma_1 =& 0.00000418328220019859\end{aligned}$$

The same process which was followed for a third order implementation can be done for a fifth order. Figure 5.38 shows the filter structure for a fifth order CRFF.

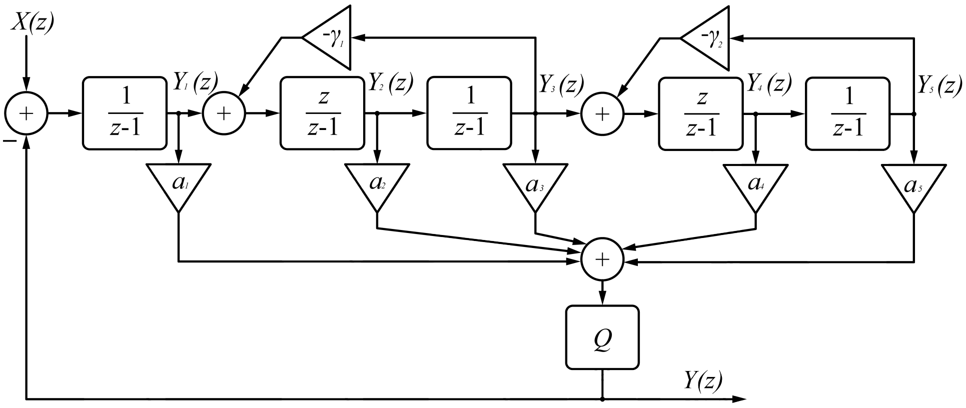


Figure 5.38: Fifth order CRFF filter adapted from Norsworthy *et al.* (1997)

The transfer function for the individual sections of the fifth order filter structure shown in Figure 5.38 is

$$\begin{aligned}\frac{Y(z)}{X(z)} =& \frac{a_1}{z-1} + \frac{a_2 z}{z^2(-2+\gamma_1)z+1} + \frac{1}{z-1} \cdot \frac{a_3 z}{z^2(-2+\gamma_1)z+1}\\ &+ \frac{a_4 z}{z^2(-2+\gamma_2)z+1} \cdot \frac{z}{z^2(-2+\gamma_1)z+1}\\ &+ \frac{1}{z-1} \cdot \frac{z}{z^2(-2+\gamma_1)z+1} \cdot \frac{a_5 z}{z^2(-2+\gamma_2)z+1}\end{aligned} \qquad (5.6.12)$$

When the individual sections are combined, they form the transfer function

$$\frac{Y(z)}{X(z)} = \frac{a_1 - 4a_1z - a_2z + a_3z + 6a_1z^2 - 4a_1z^3 + a_1z^4 + 3a_2z^2 - 3a_2z^3 + a_2z^4 - 2a_3z^2}{5z - \gamma_1z - \gamma_2z + 3\gamma_1z^2 - 3\gamma_1z^3 + \gamma_1z^4 + 3\gamma_2z^2} \cdots$$

$$\frac{+a_3z^3 - a_4z^2 + a_4z^3 + a_5z^2 - 2a_1\gamma_1z^2 + a_1\gamma_1z^3 - 2a_1\gamma_2z^2 + a_1\gamma_2z^3 - a_2\gamma_2z^2}{-3\gamma_2z^3 + \gamma_2z^4 - 10z^2 + 10z^3 - 5z^4 + z^5} \cdots$$

$$\frac{+a_2\gamma_2z^3 + a_3\gamma_2z^2 + a_1\gamma_1z + a_1\gamma_2z + a_1\gamma_1\gamma_2z^2}{-\gamma_1\gamma_2z^2 + \gamma_1\gamma_2z^3 - 1}. \tag{5.6.13}$$

Combining the transfer function of the filter structure in terms of z yields

$$\frac{Y(z)}{X(z)} = \frac{(a_1 + a_2)z^4 + (a_3 - 3a_2 - 4a_1 + a_4 + a_1\gamma_1 + a_1\gamma_2 + a_2\gamma_2)z^3}{z^5 + (\gamma_1 + \gamma_2 - 5)z^4 + (\gamma_1\gamma_2 - 3\gamma_2 - 3\gamma_1 + 10)z^3} \cdots$$

$$\frac{+(6a_1 + 3a_2 - 2a_3 - a_4 + a_5 - 2a_1\gamma_1 - 2a_1\gamma_2 - a_2\gamma_2 + a_3\gamma_2 + a_1\gamma_1\gamma_2)z^2}{+(3\gamma_1 + 3\gamma_2 - \gamma_1\gamma_2 - 10)z^2} \cdots$$

$$\frac{+(a_3 - a_2 - 4a_1 + a_1\gamma_1 + a_1\gamma_2)z + a_1}{+(5 - \gamma_2 - \gamma_1)z - 1}. \tag{5.6.14}$$

The loop filter transfer function which was exported from Matlab® is

$$\frac{b(z)}{a(z)} = \frac{0.012097794834141z^4 - 0.048238085270980z^3 + 0.072128453864227z^2}{z^5 - 4.999994897539333z^4 + 9.999984692621844z^3} \cdots$$

$$\frac{-0.047933828259711z + 0.011945664836945}{-9.999984692621844z^2 + 4.999994897539333z - 1}. \tag{5.6.15}$$

Equating Equation 5.6.14 with Equation 5.6.15 yields the coefficients for the fifth order filter as being

$$\begin{aligned}
\mathbf{a_1} &= 0.011945664836945 \\
\mathbf{a_2} &= 1.521299971961772 \times 10^{-4} \\
\mathbf{a_3} &= 9.001329807155925 \times 10^{-7} \\
\mathbf{a_4} &= 2.843253341138510 \times 10^{-9} \\
\mathbf{a_5} &= 3.748943391513047 \times 10^{-12} \\
\gamma_1 &= 4.183047168858502 \times 10^{-6} \\
\gamma_2 &= 9.194134984581749 \times 10^{-7}
\end{aligned}$$

The filter coefficients can then be simulated by forming a filter model in Simulink which would replace the transfer function which was previously being used. This would approximate the real world implementation of the noise shaper. Figure 5.39 shows the Simulink model for a third order loop filter while Figure 5.40 shows the fifth order equivalent.

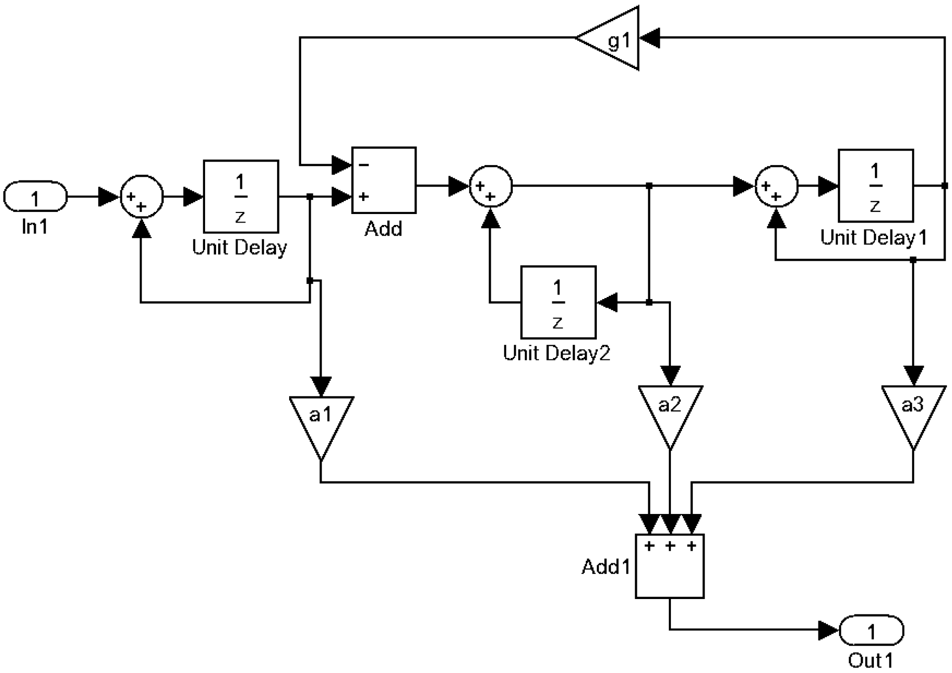


Figure 5.39: Third order loop filter Simulink implementation

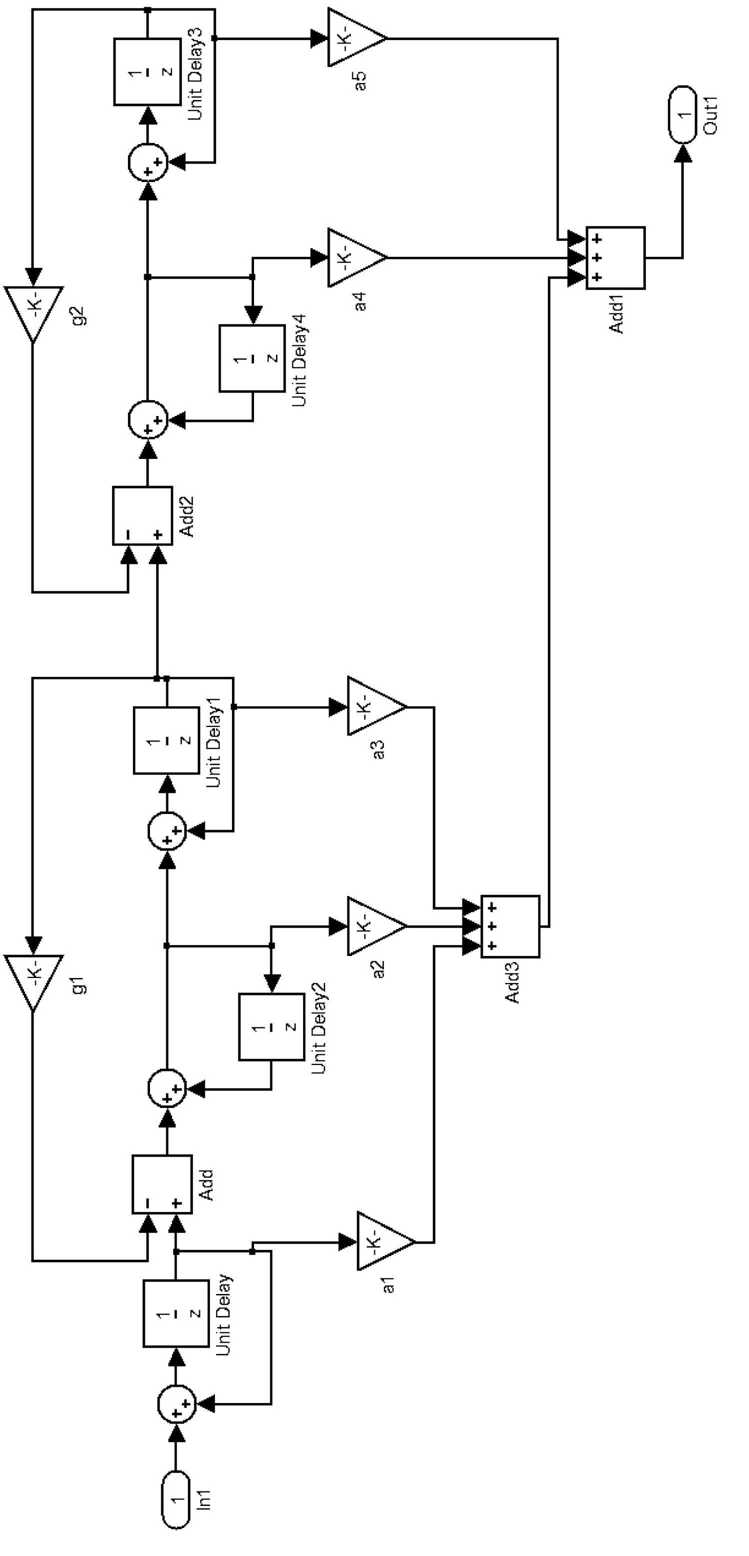


Figure 5.40: Fifth order loop filter Simulink implementation

The results obtained when using the calculated filter coefficients and when using the transfer function exported from SISOTOOL were equivalent. This verifies that the coefficients are viable.

5.7 Conclusion

Noise shaping presented an effective way of achieving high resolution audio with a low resolution modulator. There were two categories which were investigated. The most obvious difference between them was the placement of their primary noise shaping filter.

The loop filter approach proved to be far more effective than the general noise shaper technique discussed in Section 5.2. This is especially true when ripple compensation was coupled with the loop filter.

When viewing the various loop filter configurations, the one which stood out for practical implementation was the CRFF structure. It made it possible to place poles above DC and it would not require a large amount of system resources when implemented on programmable logic.

Chapter 6

VHDL Implementation

Very high speed integrated circuits (V) Hardware Design Language (HDL), VHDL, was the chosen format for developing this project's firmware. The theory behind the various blocks required for an effective PCM to PWM conversion were detailed in previous chapters. This chapter will outline their implementation in a system which could be used in practice.

The first consideration for implementation was that the data needs to be processed in real-time as new audio data would be constantly presented to the audio processor. The various firmware blocks presented is this chapter are the device configuration, audio receiver, interpolation, feedback, noise shaper and the pulse width modulator. VHDL allows for subprograms to be created and then linked together. This feature allows for an easier programming environment where pipelined data processing becomes simplified (Ashenden, 2008).

An overview of a complete audio system will be shown which illustrates how the FPGA would fit into a real life situation. Continuing from the overview, a more detailed block diagram of the various subsections within the FPGA will be shown which illustrates their interconnectivity.

This chapter will conclude with a detailing of the resources required to implement such a system on an FPGA and how the various sub systems are synchronised with each other.

6.1 Hardware interconnections

The block diagram in Figure 6.1 shows a complete audio amplifier system with a digital audio input and an amplified analogue waveform driving a loudspeaker.

The first block is the digital audio source, which will transmit data using either the AES/EBU or S/PDIF standard. The audio data is received from the audio source by the sample rate converter with integrated digital audio receiver device, in this case the SRC4392 from Texas Instruments. Digital audio data with either 16, 18, 20 or 24 bits of resolution with a sample rate from 20 kHz to 216 kHz is acceptable. The SRC4392 device then up-samples the audio data to a user defined sample rate, below 216 kHz, and passes this new information to the FPGA in PCM format. The FPGA chosen was the Cyclone III (EP3C25) from Altera®.

The FPGA performs the PCM to PWM conversion while taking into account the nonlinearities and low resolution audio normally associated with such a conversion. The pulse width modulated signal then drives the class-D switching amplifier's input with a switching frequency of 384 kHz.

The class-D power stage effectively amplifies the waveform which comes from the FPGA. The amplified pulses are then filtered by the amplifier's passive low-pass filter which attenuates the carrier. This filtered waveform is akin to the original audio only as an analogue version. The loudspeaker is then driven by the analogue waveform which finally reproduces the original program material which is typically recorded from an analogue source, as most music is. This system can therefore be thought of as a high powered digital to analogue converter.

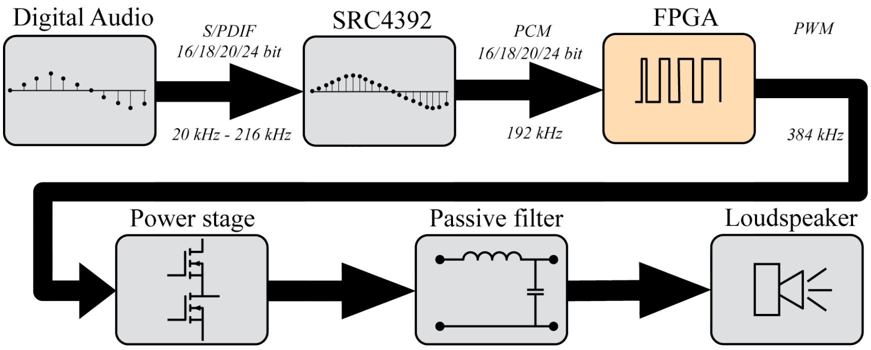


Figure 6.1: Block diagram of a complete audio system

Shown in Figure 6.2 is a block diagram of the interconnections between the firmware modules within the FPGA. When the FPGA is powered on, configuration of the internal Phase Lock Loop (PLL) takes place. Once this is complete, the SRC4392 device is configured via the Serial Peripheral Interface (SPI) bus.

Audio data is then received by the FPGA in a serial data stream and stored as data words which are used for processing. The received audio is used to generate new data samples by the interpolation block.

Negative feedback and a loop filter are used for the noise shaping section which then feeds a reference to the PWM block.

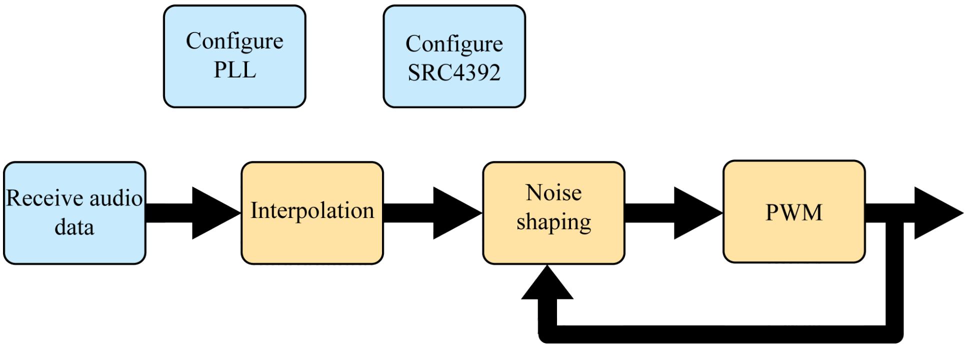


Figure 6.2: Firmware block diagram

6.2 Configuration

Each of the various subsystems shown in Figure 6.2 requires a specific clock frequency for operation within the whole system. Certain blocks run at faster or slower speeds than others depending on their function. The desire is that they are synchronised with each other and complete their operations at the same time.

The main system clock is provided by an external oscillator which emits a 24.576 MHz clock pulse. The internal PLL of the Cyclone III is then used to generate the remaining clock frequencies required from this main system clock. The megafunction wizard of Altera®'s Quartus II software was utilised to configure the PLL. Table 6.1 lists the different clocks which were generated by the PLL. It is interesting to note that they are all multiples of 48 kHz.

Table 6.1: PLL clock generation

Clock speed	Description
24.576 MHz	Master system clock used as PLL input
49.152 MHz	Faster clock used for most system blocks
12.288 MHz	Audio data bit clock, connected to SRC4392
192 kHz	Audio word clock, connected to SRC4392

The specific distribution of the clock signals is shown in Figure 6.3. The 49.152 MHz is used predominately for what could be deemed the calculation blocks, and the slower 24.576 MHz clock is used for the blocks where information is updated less often.

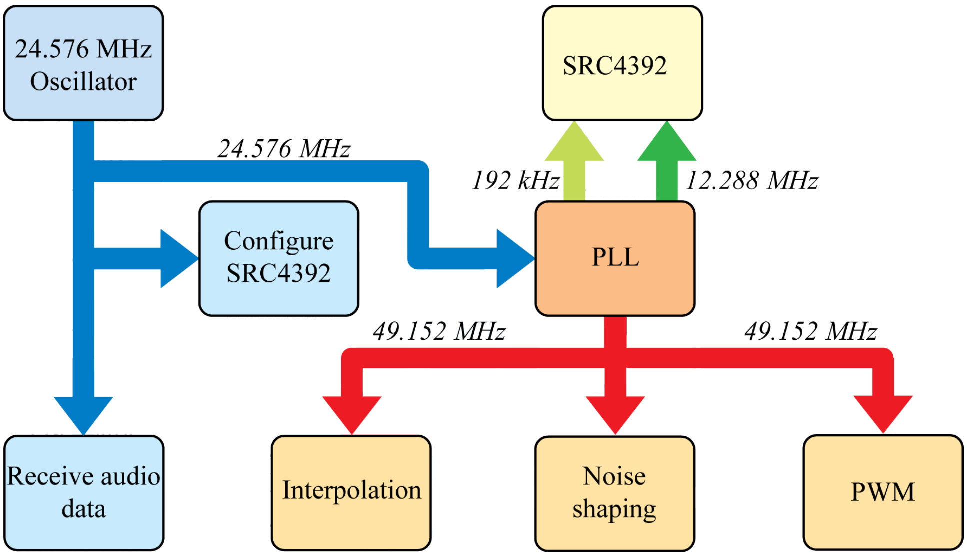


Figure 6.3: PLL clock distribution

The next step was to configure the SRC4392 device via an SPI interface. Specific data values needed to be sent to the SRC4392's registers as there are many different configuration options available. The audio receiver's data registers were configured for the following internal functions to take place:

- Enable sample rate converter function block,
- enable receiver function block,
- enable serial port A,

- configure port A for 24-bit right justified data,
- put SRC4392 into slave mode,
- output data source of port A is the internal SRC block,
- receive audio data from control register RX3,
- automatically mute output if a loss of lock is detected,
- configure the internal PLL for a 24.576 MHz clock,
- SRC input data source is the internal Digital Interface Receiver,
- enable output attenuation tracking,
- SRC output word length set to 24 bits and
- set output attenuation to −2 dB.

The SPI interface required the use of five of the FPGA's I/O pins . These were for the:

- Reset (active low), RST,
- chip select (active low), CS,
- master clock, MCLK,
- serial data clock, CCLK, and
- data in, which is clocked on the rising edge of CCLK, CDIN.

Three bytes of data were required to program each register. The first byte contained the register address and whether is was being read from or written to. The second byte is comprised of zeros, as it is just there for when 16 bit addressing is used. The third byte of information contains the register data. Data being written to the device is clocked in on the rising edge of the serial data clock.

The option of reading back information from the registers was not established in this instance, yet it is possible. Material being read from the SRC4392 is clocked out on the falling edge of the serial data clock.

Figure 6.4 shows the SPI interface with regards to clock timing. The CS pin goes low, which informs the SRC4392 device that the SPI is active. The CCLK pin clocks in data from the CDIN pin, which streams the information in serially. The zoomed view of the first byte of CDIN shows that the most significant bit lets the slave device know if the master device is reading or writing to a particular register address.

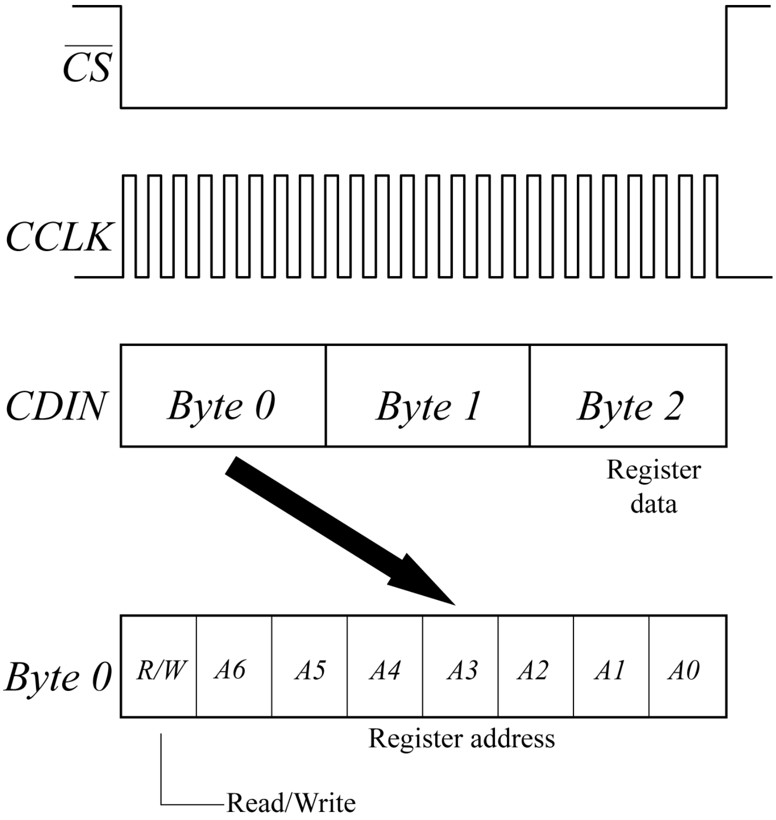


Figure 6.4: Configuration via SPI

The state diagram shown in Figure 6.5 illustrates the process by which data is sent to the SRC4392 via SPI.

The natural resting position for the SPI process is to be in idle mode. At this point, the CS pin is held high, the CCLK pin is held low and the CDIN pin is held low. The data which will be transmitted is held in an array, which is called through an address system activated by a counter.

The program then moves to the load data state. The CS pin, which selects the slave device, goes low. The bit which corresponds to a running counter

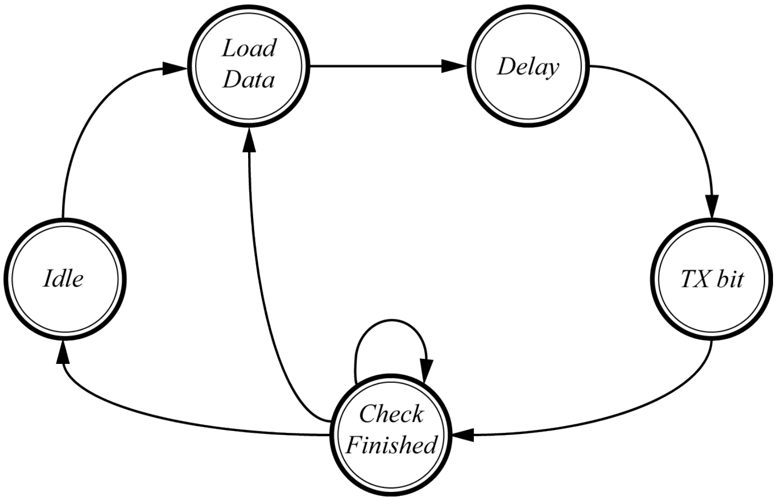


Figure 6.5: SPI state diagram

value, which acts as the index, is directed to the CDIN pin. The CCLK is again forced to a low position, which becomes relevant when the program loops back.

The next state delays the process by one clock cycle. This ensures enough time between sending serial data out to maintain a reliable transmission baud rate. The TX bit state is then activated. The only function which is carried out is to change the CCLK pin to a high, which then forms the clocking characteristic of this pin.

The check finished state verifies whether all the bits of a word have been transmitted, if they have, then the program moves onto a new word to be transmitted. If all the words have been sent, then the program remains in the idle mode. Section B.2 in the Appendix lists the VHDL code for this process.

6.3 Receive audio data

The conversion of serial data to parallel is carried out as new audio material is presented to the FPGA by the SRC4392 device. Information is clocked into a register of the FPGA and when a complete word is received, the data is carried forward for further processing.

Two loops are used for the sample collection. When the left/right clock goes high, it signifies that left channel data is being sent. The first loop activates on the left channel transmission. The second loop checks for the bit clock to pulse, whenever it goes high, a single bit of data is clocked into a buffer. The buffer shifts its data through until the left/right clock switches to the right channel, signifying the completion of the left channel data.

Another flag which is activated when there is a switch between left and right channels, is to signify the completion of receiving the audio word so that the next VHDL block can begin running.

Figure 6.6 shows right-justified data being synchronised with the bit clock and left/right clock.

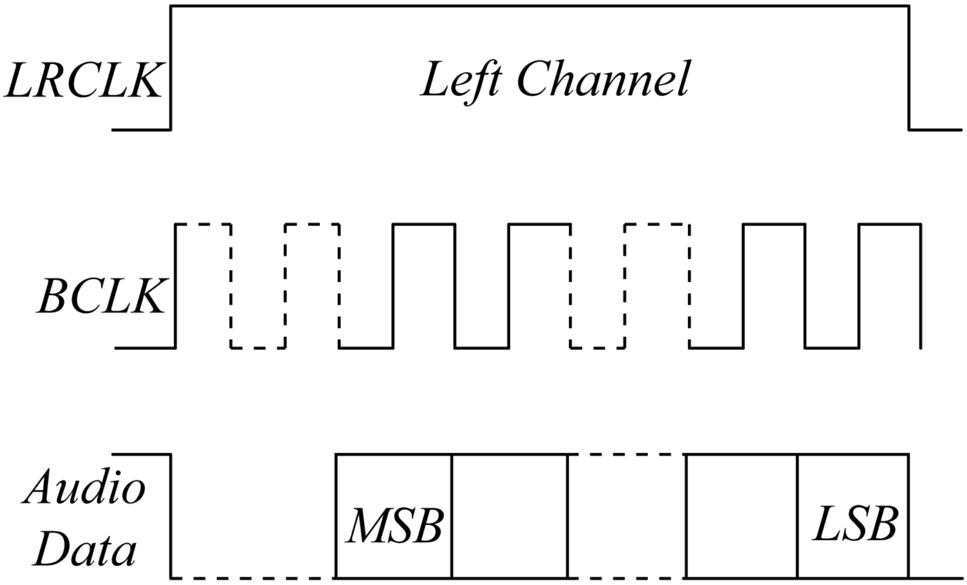


Figure 6.6: Right-justified data format

The next processing block waits for the completion of the audio receiver, then stores the received data in a buffer. The buffer contains six pieces of data. They are required for fifth order polynomial interpolation. The state diagram for the audio buffer is shown in Figure 6.7.

When in idle mode, the program waits until a flag from the previous VHDL block has been activated. This signifies that a new audio sample has been received.

The accept data state shifts the buffer data within an array, where the oldest data word is discarded. The next state moves the newest information into the array before going to the succeeding state. The data out state activates the following VHDL program block, which is part of the interpolation section. This signifies that the new data has been placed into the array and is ready

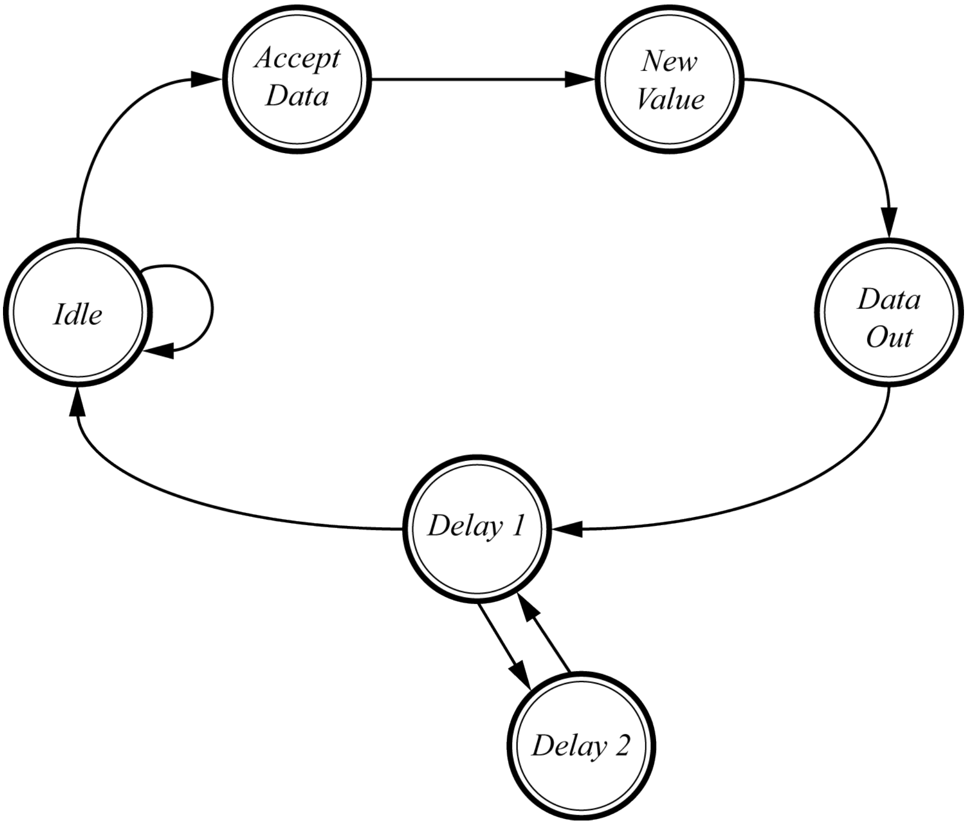


Figure 6.7: Audio buffer state diagram

to be used. The six buffered values are made available globally to the other processing blocks of the system.

The delay states are there to ensure that the interpolation block is not constantly activated. This state waits until the complete receive flag of the previous block is disabled before it allows the state to move into the idle mode.

6.4 Interpolation

The interpolation section was divided into two VHDL subsystems. The first generates polynomial coefficients using the six data values stored in the audio buffer. The second calculates new data points based on the coefficients.

The polynomial coefficients are determined by using the same method presented in section 4.6.2, Newton's interpolation formula. Figure 6.8 shows the state diagram for the coefficient calculations. Each state forms part of the divided differences table, with the next state requiring the calculated values of the last. The VHDL source code which describes this process is available in section B.4 of the Appendix.

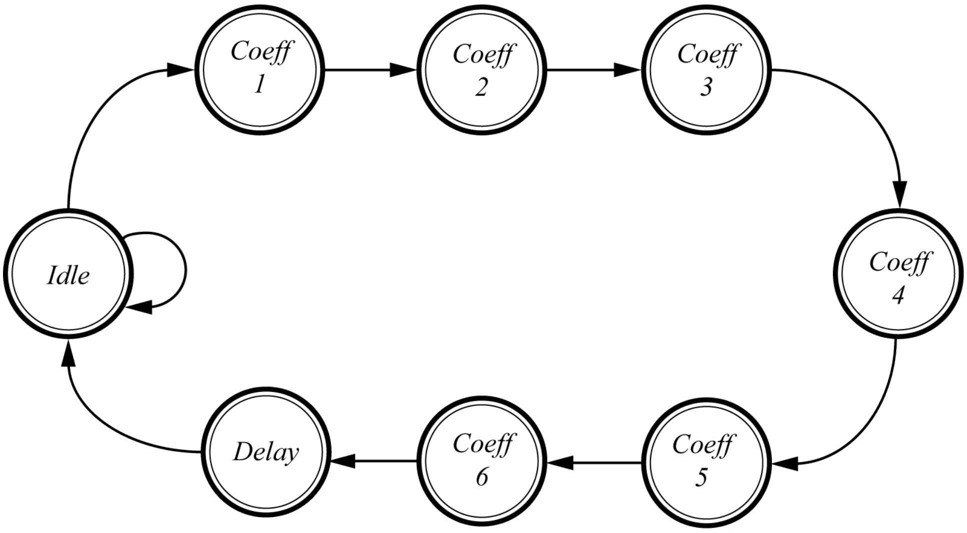


Figure 6.8: Polynomial coefficients state diagram

Once the coefficients have been determined, the polynomial calculation block is activated. In order to economise FPGA resources, part of the calculation is performed offline using Matlab®. The results are stored in five Read Only Memory (ROM) blocks. There are 64 memory locations in each of the five ROM blocks, this relates to there being 64 new data samples being generated for every single audio word being received from the SRC4392. The format chosen for the memory storage was that of a Memory Initialisation File (MIF). The state diagram for the polynomial calculation block is shown in Figure 6.9.

The process for the polynomial calculation begins in a similar manner to that of many of the other processing blocks. The first state is an idle state which is held until there is a notification that the polynomial coefficients have been calculated. Once the signal has been received that the coefficients are complete, the coefficients are first converted from signed values to the signed fixed point format. They were previously kept as just signed values to reduce resource usage.

The next step of the state machine is to calculate a new data value using the coefficients and values from the ROM blocks, addition and multiplication

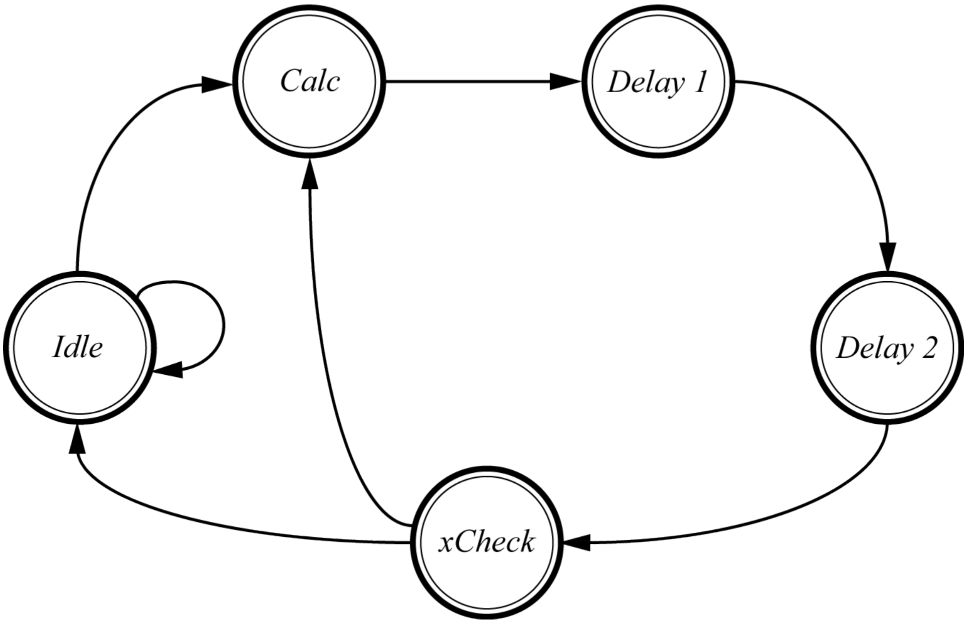


Figure 6.9: Polynomial calculation state diagram

of values is carried out. The address from which to locate the required value from each of the ROM blocks is then incremented.

The next two delay states provide the two clock cycles necessary for the previous multiplication to complete. The xCheck state provides a path for the newly calculated data sample to move onto the other blocks of the global system. The position of the address counter for the ROM blocks is then checked. If the end of the memory table has been reached, the address is reset to zero and the whole process begins afresh.

The 64 times up-sampling relates to a change in the sample rate from 192 kHz to 12.288 MHz. This process is illustrated in Figure 6.10, the 49.152 MHz clock which is used as the interpolation subsystem clock is included in the diagram.

6.5 Noise shaping

By choosing the CRFF structured loop filter, implementation is less demanding than if a structure with more filter coefficients and multipliers were used. The feedback and noise shaping are both synchronised to the 49.152 MHz clock which is depicted in Figure 6.10.

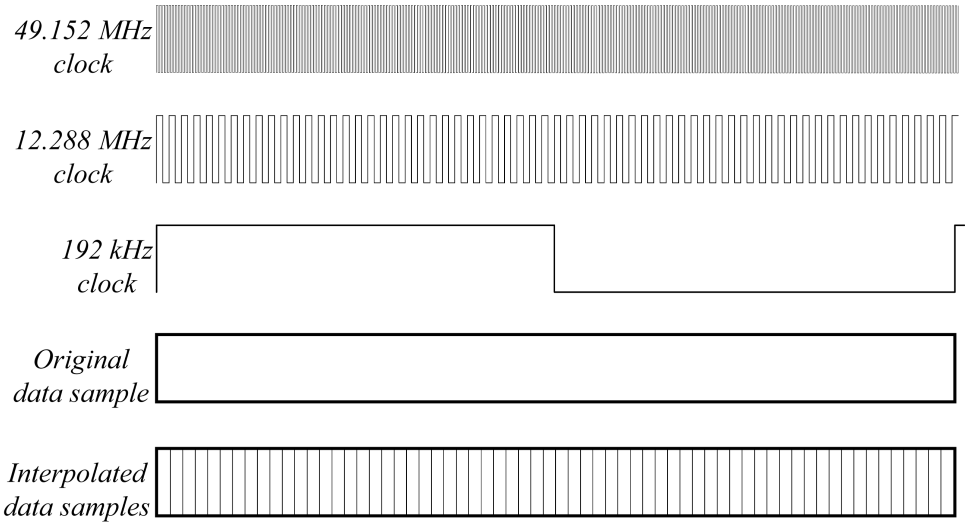


Figure 6.10: Timing diagram for the sample rate conversion

The important part of the loop filter design is to make sure that the registers never overflow. They therefore needed to be large enough to handle any typical audio waveform data which has been integrated by the loop filter. Limits for the register sizes were established by using the Simulink model of the loop filter configuration as shown in Figure 5.26. A sine wave with a frequency of 20 kHz and an amplitude of 0.9 times full scale was employed. This provided a good example of the maximum and minimum word lengths that would be required for stable operation. The largest of the registers had an integer part of 40 bits with a fractional part of 31 bits. This was for the final section of the loop filter.

The approach taken to calculate the loop filter result was very similar to the approach taken for all the other firmware blocks. A state machine was configured which divided the process up into steps. The final result is a combination of delayed and non-delayed versions of the various registers. The following calculations took place within the state machine to allow for the filter's output to be calculated:

- $Y_1 = \text{Filter input} + Y_{1(delay)}$
- $Y_2 = Y_{1(delay)} + Y_{2(delay)} - (\gamma_1 \times Y_{3(delay)})$
- $Y_3 = Y_2 + Y_{3(delay)}$
- $Y_4 = Y_{4(delay)} + Y_{3(delay)} - (\gamma_2 \times Y_{5(delay)})$

- $Y_5 = Y_4 + Y_{5(delay)}$
- Filter output $= (Y_{1(delay)} \times a_1) + (Y_2 \times a_2) + (Y_{3(delay)} \times a_3) + (Y_4 \times a_4) + (Y_{5(delay)} \times a_5)$

The state machine took four clock cycles to complete, this was due to delays being added which allowed for the multiplications to settle.

6.6 Pulse width modulation and feedback

The PWM block utilised the 49.152 MHz clock as its deciding factor for operation. The system clock directly affects the switching frequency of the pulse width modulator. The relationship between the PWM resolution, switching frequency and system clock frequency is described by

$$\text{Switching frequency} = \frac{\text{System clock frequency}}{2^{\text{PWM bit resolution}}}$$

$$f_{sw} = \frac{49.152\ MHz}{2^7} = \frac{49152000}{128} \tag{6.6.1}$$

$$=384\ kHz.$$

The sawtooth waveform was generated by utilising a 7 bit counter which counts up from its minimum to maximum, then resets and repeats. The counter had a signed value which went from -1 to 0.9923. The comparison between the sawtooth and the value coming from the filter was made each time the sawtooth counter incremented.

If the value of the sawtooth was larger than that of the audio, the PWM output was set to a logical zero, and vice-versa if the audio had a larger value. The feedback was determined from the same comparison. The difference was that instead of the PWM feedback value being zero, it would be negative one. Ripple compensation was included at this point. The value of the sawtooth was added to the PWM's positive or negative value to complete the compensation.

The feedback loop is constructed by subtracting the value leaving the PWM block and the value leaving the interpolation block. There is a subtraction between the data because negative feedback is being used.

6.7 Synchronisation

Synchronisation between the blocks was possible due to the fact that the various clocks used are all multiples of one another. The 24.576 MHz oscillator which was used is a multiple of 48 kHz, which as previously mentioned is a common audio sampling rate. The audio data from the SRC is upsampled to 192 kHz regardless of whether it is 32 kHz, 44.1 kHz, 48 kHz or 96 kHz. Therefore a constant 192 kHz bit stream is sent to the FPGA for processing, which is again a multiple of 48 kHz.

6.8 Conclusion

This chapter described the various firmware blocks which were implemented.

The various blocks had a common design which was the use of state machines. Certain parts of the system could be sequentially executed while the main blocks ran in parallel with each other. This makes the real-time processing possible. Being a real-time system it is important to make sure that timing deadlines are met.

The system resources required for the system implementation are shown in Table 6.2. As can be seen from the total available resources, many more features can still be added to the firmware.

Table 6.2: Required FPGA system resources

Family	Cyclone III
Device	EP3C25Q240C8
Total logic elements	5470 / 24624 (22%)
Total combinational function	4787 / 24624 (19%)
Dedicated logic registers	1650 / 24624 (7%)
Total pins	10 / 149 (7%)
Total memory bits	9600 / 608256 (2%)
Embedded multiplier 9-bit elements	97 / 132 (73%)
Total PLLs	1/4 (25%)

Chapter 7

Test and verification

This chapter outlines the problems which were encountered during the hardware implementation of the project. Due to unforeseeable circumstances, additional design work was required to ensure operability of the system. Fault finding was carried out as per any developing system and errors were found and corrected for. The information detailed in this chapter extend back to a redesign and implementation primarily of the noise shaper. This in turn affected a variation of the feedback and pulse-width modulator parts of the project.

This chapter merely details the various deviations from the original path of the project with respect to the VHDL implementation. What worked in the simulation was not as straight forward to equip in firmware. The final results will be shown in Chapter 8.

7.1 Initial testing

The first tests with the firmware running on the FPGA were as expected. The pulse-width modulator produced a pulse train which varied according to the input reference waveform. The interpolation part of the system was enabled and behaved as expected. When the noise shaper was introduced it did not behave as expected.

Much time was spent finding faults in the VHDL code. One fault was in the latency of the noise shaping filter. Due to restrictions placed on how multiplications are handled by the FPGA additional delays were required within the filter. The reason for this is that once a multiplication has taken place, a clock cycle needs to pass before the result of said multiplication can be used. The clock cycle delays which were introduced result in new data samples being produced by the filter at a rate eight times slower than required.

The firmware was modified and made as compact as possible, the reduced latency showed a significant improvement in flattening the frequency response of the system, yet the noise floor did not improve at all. This was due to the filter generating new output samples at a rate which was four times slower than required. Again, further redesign took place and the clock which activates the filter block was increased.

The filter was initially running with a clock rate of 49.152 MHz. This provided a stable platform for calculations to take place and produced untainted results. By increasing the clock rate of the filter, it would in theory produce new samples at a rate which would coincide with the rate at which the PWM was running. What was found is that the FPGA becomes unstable when performing large calculations with a high clock rate. Glitches were produced when even a 98.304 MHz clock was used, errors would occur randomly in the multiplication between filter coefficients and the data passing through the filter.

This situation caused a paradox in the research. If the clock rate was kept at 49.152 MHz then the filter would not operate as expected, due to new samples being produced too rarely, and noise shaping would not take place. If the clock rate of the filter was increased, then errors would occur causing the results to be unusable and therefore noise shaping would not occur.

7.2 System redesign

It was found through simulation that the filter needed to operate at the same rate as the pulse-width modulator. This was due to the feedback. If the PWM signal being fed back to the filter was sampled at a lower rate than the PWM frequency, then the feedback result when the PWM waveform switched between positive and negative became ambiguous.

The solution to solving the problem of the ambiguous feedback signal was to average the samples coming from the pulse-width modulator, before ripple compensation takes place. Introducing a running average filter into the feedback path would allow for a filter to be introduced which operated at a lower rate. The filter was redesigned to operate at 12.288 MHz which allowed for a 49.152 MHz system clock to be used and ensured stability in the calculations. Figure 7.1 shows the modified Simulink model which incorporates the running average filter. Four samples are used in the average calculation which allows for a filter which operates four times slower than the PWM block.

The filter coefficients were calculated using the same method described in Chapter 5. The coefficients do change depending on the sample rate used, even though the positions of the poles and zeros of the filter do not.

The new coefficients for the third order filter became

$$\begin{aligned}
\mathbf{a_1} &= 32.007357963907 \times 10^{-3} \\
\mathbf{a_2} &= 2.23532237809242 \times 10^{-3} \\
\mathbf{a_3} &= 37.2506773734498 \times 10^{-6} \\
\gamma_\mathbf{1} &= 14.7068533831174 \times 10^{-6}\ .
\end{aligned}$$

The fifth order order coefficients were recalculated as being

$$\begin{aligned}
\mathbf{a_1} &= 48.0667543933888 \times 10^{-3} \\
\mathbf{a_2} &= 2.49572438876488 \times 10^{-3} \\
\mathbf{a_3} &= 59.6921688489462 \times 10^{-6} \\
\mathbf{a_4} &= 754.866410865662 \times 10^{-9} \\
\mathbf{a_5} &= 4.15228295977656 \times 10^{-9} \\
\gamma_\mathbf{1} &= 52.7991875389918 \times 10^{-6} \\
\gamma_\mathbf{2} &= 11.2571316734770 \times 10^{-6}\ .
\end{aligned}$$

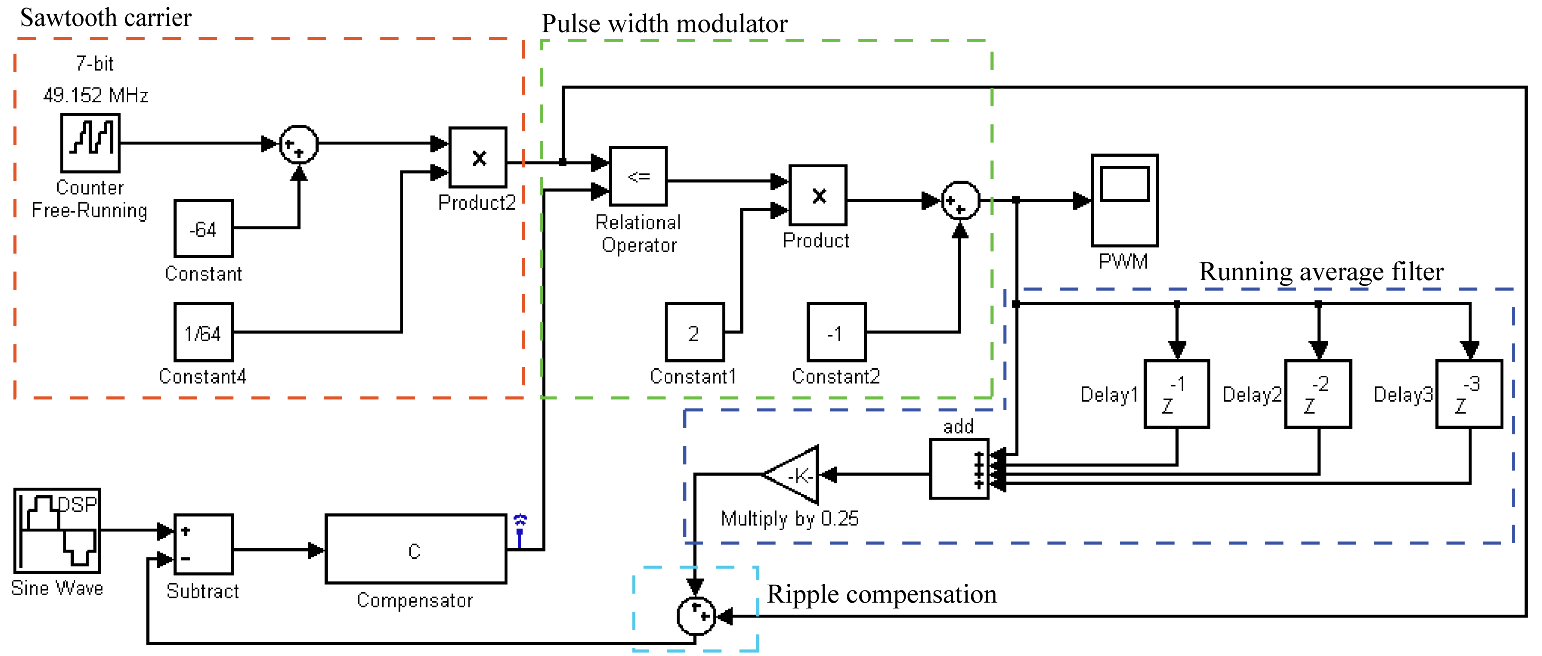


Figure 7.1: Simulink model for a pulse width modulator and loop filter including ripple compensation and PWM averaging

7.3 Conclusion

Although there is always much fault finding and reworking when new firmware is being developed, the redesign of the noise shaper proved to be a very challenging part of the project. The method for solving the problem of the filter running at a different rate to that of the PWM block is a simple one, yet difficult as it was not immediately obvious.

Chapter 8

Results

8.1 Hardware setup

The firmware was assessed using an Altera® Cyclone III based audio amplifier board which was developed by Professor H du T Mouton of Stellenbosch University, South Africa. The audio amplifier board is shown in Figure 8.1.

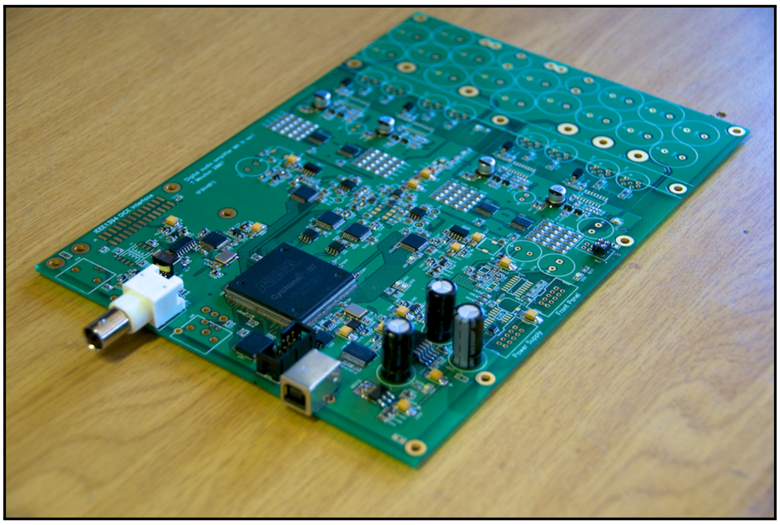


Figure 8.1: FPGA based audio amplifier board

The test instruments which were utilised include the

- Tektronix® TDS 2024B digital storage oscilloscope,
- Rohde & Schwarz® UP350 audio analyser and

- Agilent Technologies® 1683A logic analyser.

The test set up is shown in Figure 8.2. On the right is the logic analyser. The audio analyser is on the left with the oscilloscope stacked on top of it.

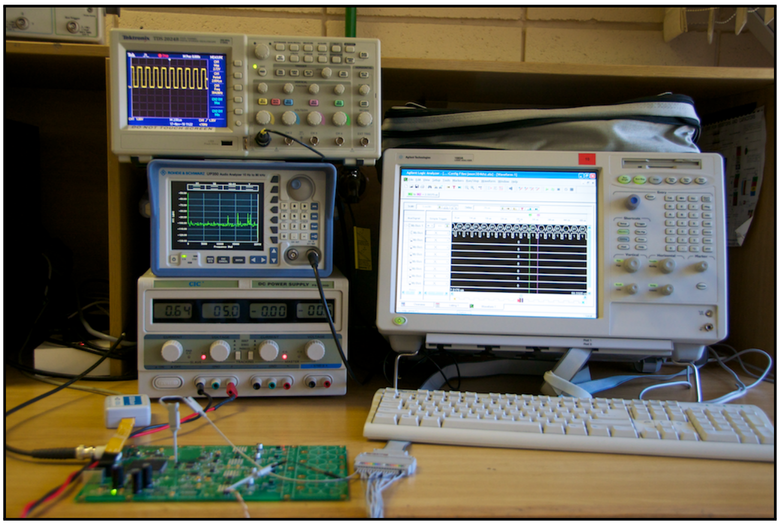


Figure 8.2: Hardware test instrument set up

8.2 Simulation verification

The VHDL development was carried out using version 9 of Altera®'s Quartus II software. In order to verify the operation of the VHDL code a comparison between the Simulink and Quartus simulation results was made. Simulating in Quartus gives a clear indication of how the VHDL code will behave once programmed onto an FPGA.

Figure 8.3 shows the input to the loop filter while Figure 8.4 shows its output. Only 1024 samples are shown due to each value from the VHDL simulation having to be entered manually into MATLAB® to produce the waveform.

The values which enter the filter's input, directly correspond to that of the Simulink simulation and the VHDL code. This verifies that the pulse width modulator, ripple compensation, running average filter and the feedback are operating as expected. The output of the compensation filter verifies that the filter calculations are being processed correctly.

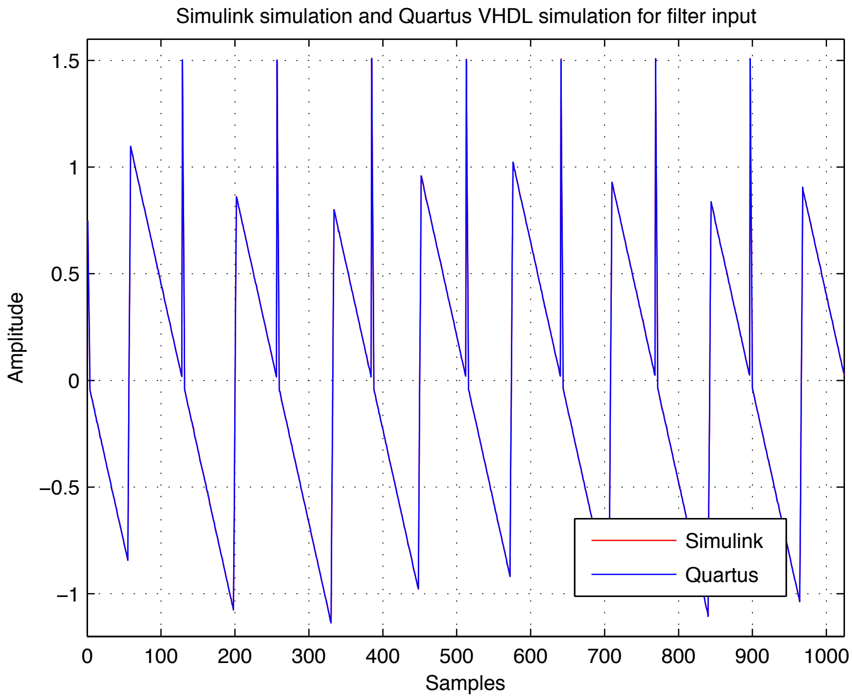


Figure 8.3: Comparisson between Simulink and Quartus simulations for the filter input

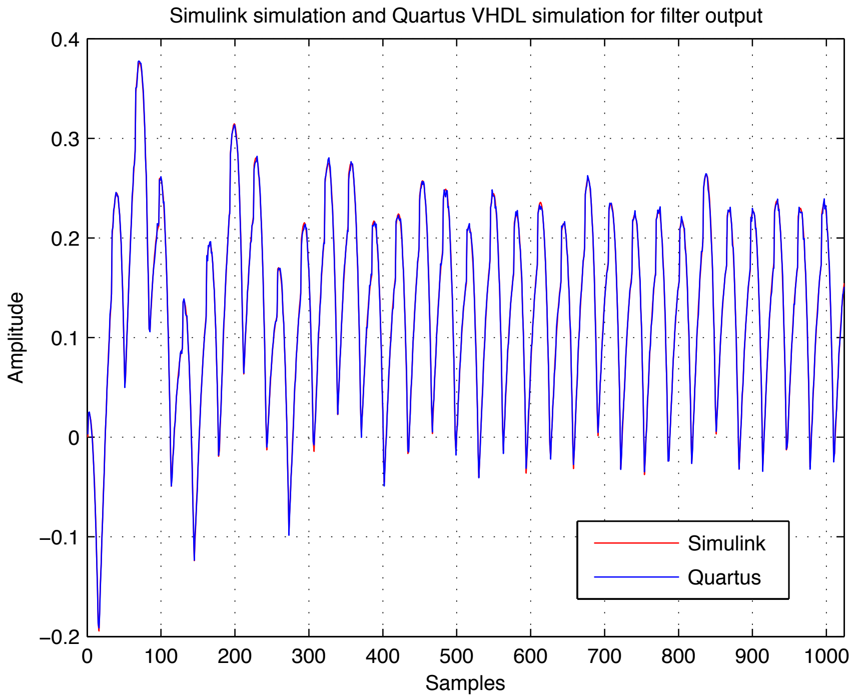


Figure 8.4: Comparisson between Simulink and Quartus simulations for the filter output

8.3 Measurements

The first of the tests were conducted using the UP350 audio analyser as a digital audio source. The FFT analysis was configured to operate with 16384 points which displays the spectrum with a resolution of less than 3 Hz. The frequency of the sine wave was varied to evaluate the system's operation over the entire audio band.

Figure 8.5 shows the resulting PWM waveform when a zero amplitude audio signal is used. The duty cycle of the PWM is therefore 50%. Figure 8.6 shows the PWM waveform when a 20 kHz sine wave with an increased amplitude is used as the audio input. This shows the varying duty cycle associated with such an input source.

The logic analyser was used to verify that there were no glitches in the PWM waveform. Errors are difficult to see on an oscilloscope unless they occur periodically. The logic analyser stores a long stream pulses and a more thorough investigation is possible.

The same 20 kHz sine wave which was used previously as the audio input was again used to give the result shown in Figure 8.7. The varying pulse widths can be seen as the amplitude of the sine wave increases and decreases.

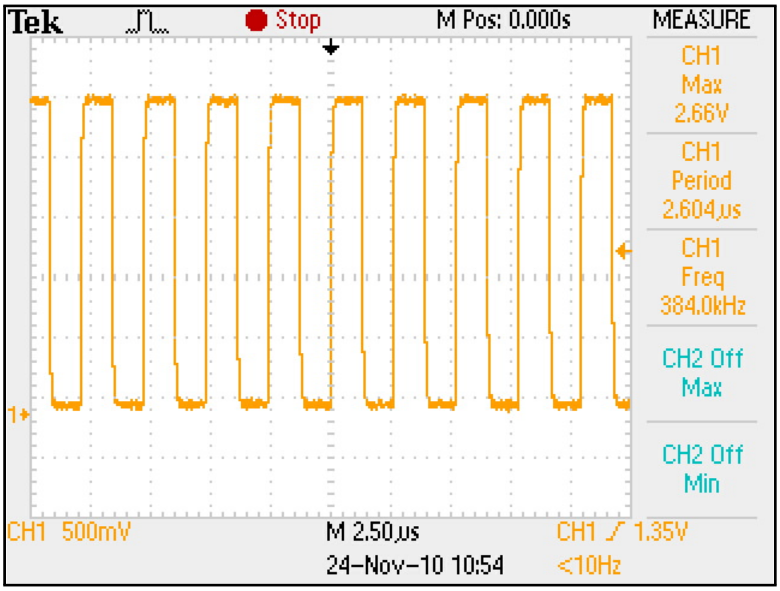


Figure 8.5: PWM waveform switching at 384 kHz with a 50% duty cycle

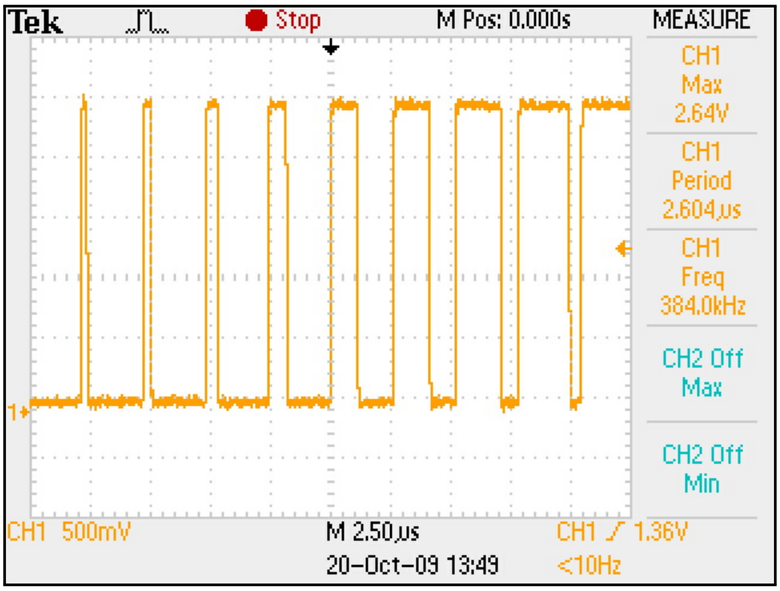


Figure 8.6: PWM waveform switching at 384 kHz modulated with a 20 kHz sine wave

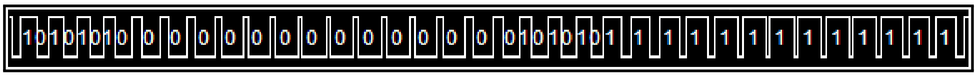


Figure 8.7: Logic analyser result of a PWM waveform switching at 384 kHz which has been modulated with a 20 kHz sine wave

Figure 8.8 shows the test set up configuration for the audio analyser when taking measurements of the spectral content of the PWM waveform. The audio analyser connects to the SRC4392 device via a coaxial cable and is electrically isolated via a transformer on the audio development board. The PCM data is then transferred to the FPGA where signal processing takes place. The PWM waveform which is present at the output port of the FPGA is then fed to a MOSFET driver, IRS20957SPBF, and then a half-bridge MOSFET, IRFI4019HG, configuration. The waveform measured at the half-bridge's mid-point is fed back to the audio analyser. The PWM waveform is not low-pass filtered as is common place when using a class-D amplifier. The reason for this is that the audio analyser band limits the spectral content to the band of interest, in this case the 22 kHz and 88 kHz bands were of interest.

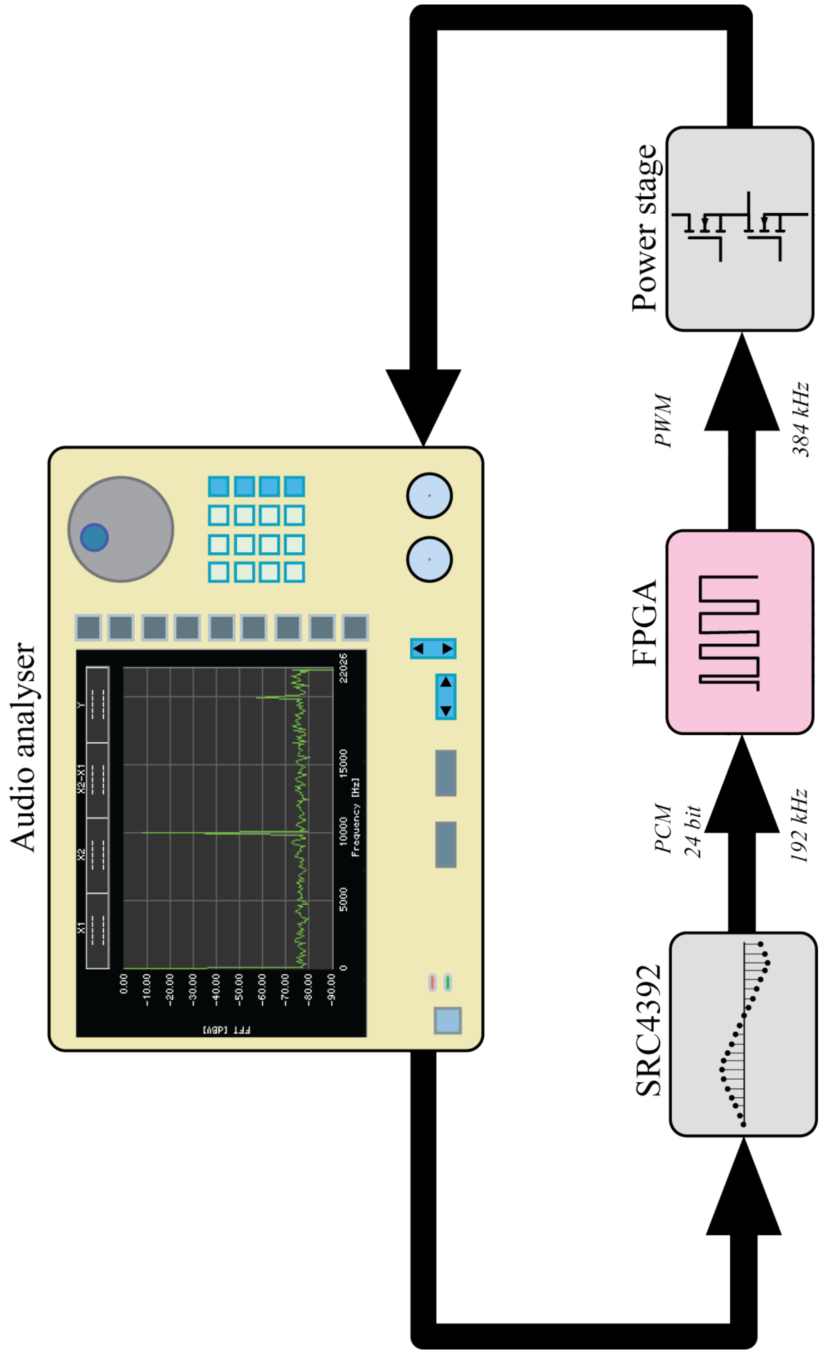


Figure 8.8: Audio analyser test set up

Figure 8.9 shows the spectral content when a 1 kHz sine wave is used as the audio input for the system. In order to demonstrate the noise-shaping action of the system, the audio analyser's viewing bandwidth was extended to 88 kHz, this is illustrated in Figure 8.10. The increasing noise floor can clearly be seen at frequencies above 20 kHz.

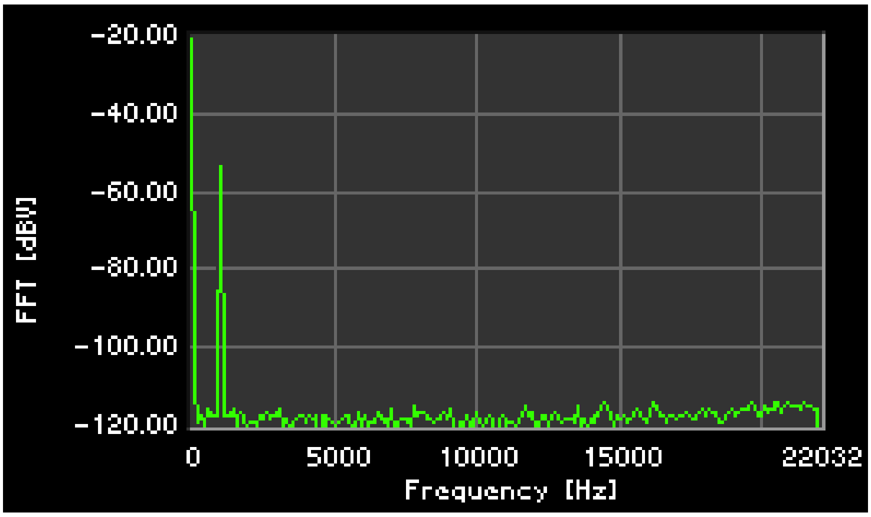


Figure 8.9: Spectral content of the PWM signal when a 1 kHz sine wave is used as the audio input, viewing the audio band

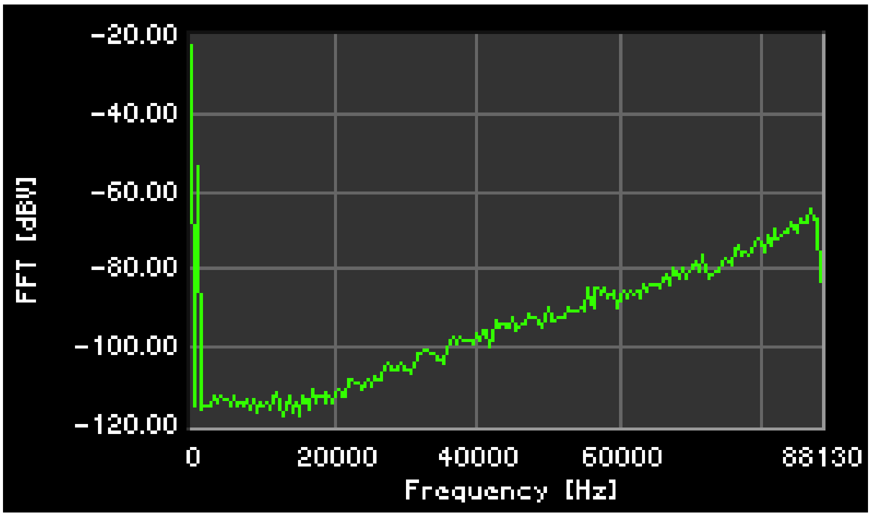


Figure 8.10: Spectral content of the PWM signal when a 1 kHz sine wave is used as the audio input, viewing a bandwidth of 88 kHz

Figure 8.11 shows the spectral content when a 10 kHz sine wave is used as the audio input for the system, while Figure 8.12 shows the same result with an 88 kHz bandwidth.

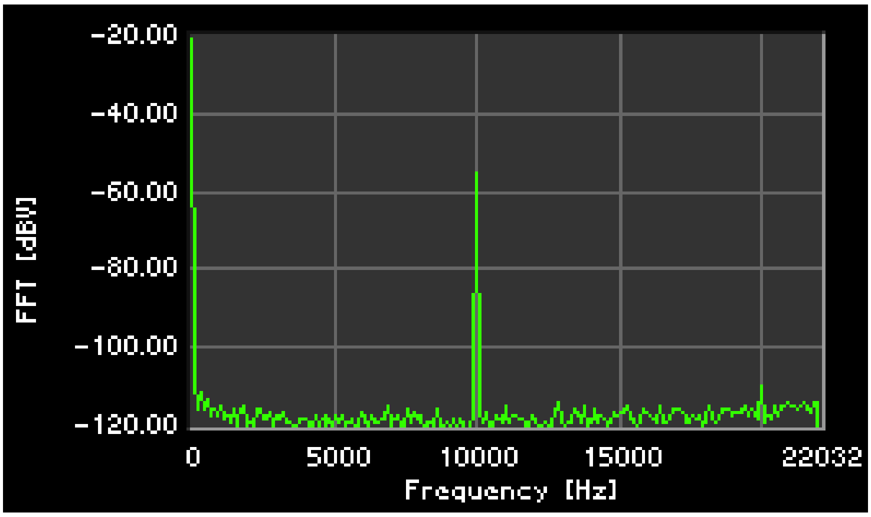


Figure 8.11: Spectral content of the PWM signal when a 10 kHz sine wave is used as the audio input, viewing the audio band

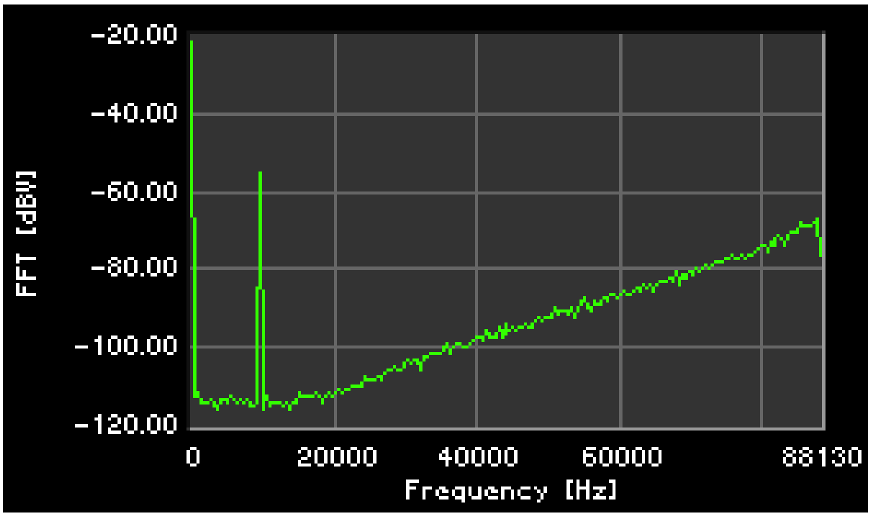


Figure 8.12: Spectral content of the PWM signal when a 10 kHz sine wave is used as the audio input, viewing a bandwidth of 88 kHz

Figure 8.13 shows the spectral content when a 20 kHz sine wave is used as the audio input, while Figure 8.14 shows the same result with a bandwidth of 88 kHz.

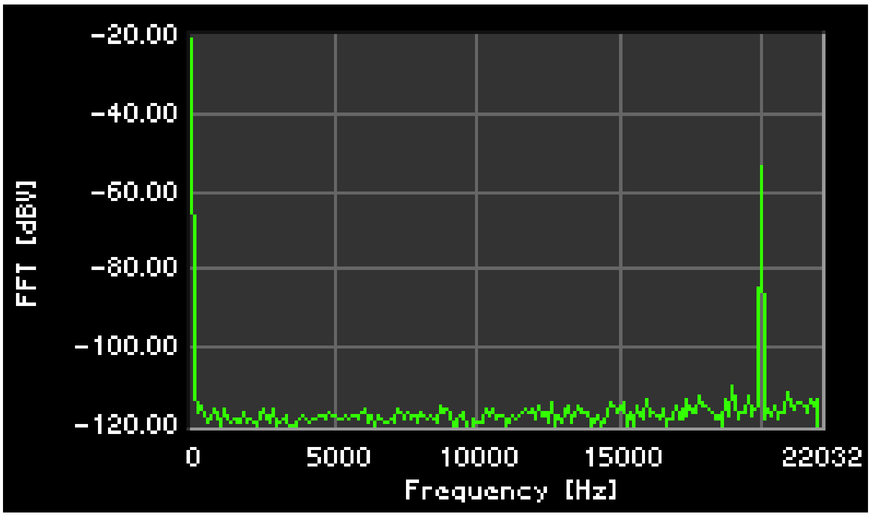


Figure 8.13: Spectral content of the PWM signal when a 20 kHz sine wave is used as the audio input, viewing the audio band

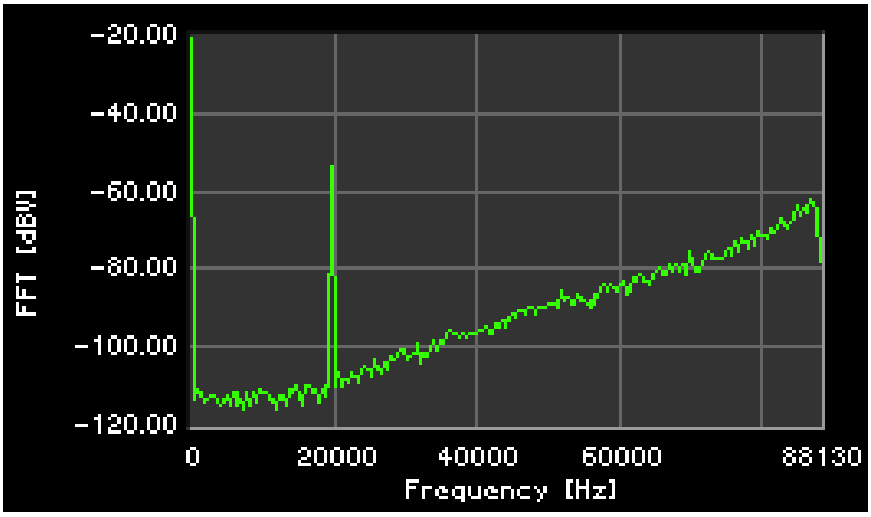


Figure 8.14: Spectral content of the PWM signal when a 20 kHz sine wave is used as the audio input, viewing a bandwidth of 88 kHz

Intermodulation (IM) distortion is generated when the nonlinear mixing of two or more frequency components occurs (Slone, 2002). When two frequency components are applied to a linear audio system, there is simply a summation. However, if the two same components are applied to a nonlinear system, a modulation of the two will occur. The two original components and the sum and difference will be present. The magnitude for which it affects a system depends on how nonlinear the audio system is. The original components will themselves be distorted proportional to the degree of modulation (Slone, 2002).

The IM distortion is a function of the linearity of the system, which is directly affected by the THD specification. If the THD is improved, the IM distortion performance improves.

Figure 8.15 shows the resulting spectrum when a two-tone input at 17 kHz and 18 kHz is utilised.

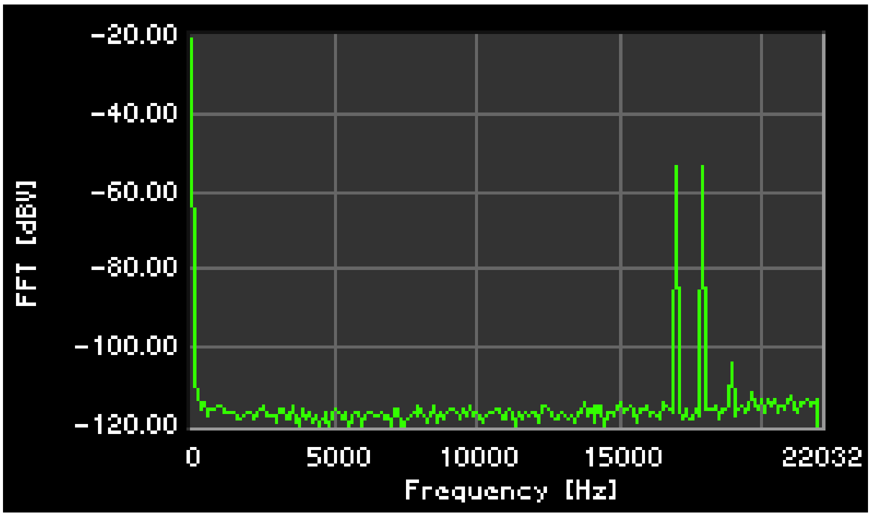


Figure 8.15: Spectral content of the PWM signal when a two-tone input at 17 kHz and 18 kHz is used as the audio input, viewing the audio band

8.4 Conclusion

Figures 8.3 and 8.4 show the comparison between the simulation results of Simulink and Quartus. There is a direct correlation of the waveforms which verifies that the calculations are being carried out as intended.

The PWM pulse train which is generated by the FPGA can be clearly seen in Figure 8.5. The 50% duty cycle of the waveform, when averaged by the low-pass filter of the class-D power stage would result in an amplitude of zero. This relates to the zero amplitude sine wave which was used as the reference.

When a 20 kHz sine wave with an increased amplitude is utilised, then the duty cycle varies quite dramatically. The PWM waveform of this particular situation is shown in Figure 8.6. When the varying pulses are averaged, the original sine wave will emerge.

Figures 8.9, 8.11 and 8.13 show the resulting spectral content of PWM waveforms when they have been modulated with 1 kHz, 10 kHz and 20 kHz sine waves respectively. The low noise floor is evident that the noise shaping action of the system is working as expected. The spectrum analyser showed a noise floor approaching -120 dB, which relates to a bit resolution in the audio band of approximately 20-bits. This performance is achieved while using only a 7-bit modulator.

Figures 8.10, 8.12 and 8.14 give a clear indication of the noise shaping action. The noise floor increases at frequencies above the audio band. This is inconsequential as those higher frequencies would be filtered out by the same low-pass filter which is used to filter out the PWM carrier.

Chapter 9

Conclusion

9.1 Overview of project objectives

The objectives which were discussed at the onset of this thesis were that of:

- Increasing the linearity of the PCM to PWM conversion process,
- reducing the system clock speed required to reconstruct a high resolution audio signal,
- simulating the various subsections,
- choosing processing schemes which are computationally efficient when implemented on embedded hardware and
- developing firmware to demonstrate a working model of the simulations.

9.2 Objectives achieved

General investigations were made into digital signal processing and pulse width modulation. The need for a digital scheme which could counteract the undesirable effects of UPWM became clear. Linearising the PWM process could help decrease the presence of harmonics in the baseband.

The issue of linearity was addressed through sample rate conversion by employing interpolation. Many different methods were presented for achieving an interpolated waveform. All of which resulted in an increase of the sample rate. Investigation of these methods found that using the Newton interpolation formula would be an economical choice. It was thought to be the most fitting for this application as many new data samples needed to be generated for each incoming audio data sample, while using a modest amount of system resources. Interpolation expanded the usable bandwidth of the system which then allowed for noise shaping.

The simulation results showed that a low noise floor in the audio band is achievable while still using a modest seven bit modulator. The loop filter topology was found to be stable and implementation in firmware was possible. The various loop filter topologies were explained with the most efficient form chosen, which used a minimal amount of multipliers.

Testing of the firmware demonstrated a functioning PWM block. When the interpolation and noise shaping subsystems were added the system remained stable. Stability is a primary concern when employing feedback as it can cause data registers to overflow and thereby result in an erratic PWM output. The system maintained its stability and produced an error free PWM waveform.

In practice, the measured results showed a low noise floor in the audio band. This result proved the effectiveness of the control scheme. Results showed that the noise floor increased at frequencies above the audio band, which is indicative of the noise shaper.

9.3 Problems encountered

Calculation errors in the loop filter were mentioned in Chapter 7. A clock rate four times faster than the rate at which new filter samples needed to be produced by the loop filter was required. The only way to eliminate the calculation errors however was to lower the clock rate. By lowering the clock rate of the filter, noise shaping no longer took place due to a mismatch between the filter and PWM feedback. The problem was solved by averaging the feedback which then eliminated the ambiguity caused by sampling the feedback at a lower rate than that at which the PWM clock was operating.

9.4 Future research and recommendations

It would be a repetitive action, yet the addition of a second audio channel to the digital system would enable a stereo reproduction.

The next step would be to utilise global feedback which requires a high precision ADC. The output of the class-D amplifier, after the analogue low-pass filter, would be fed back to the FPGA and used instead of the internally calculated feedback. The imperfections and artefacts introduced by the output power stage and filter would therefore be taken into account. This approach has been implemented by Mouton *&* Putzeys (2009) and found to be very successful. An extremely low level of distortion is achievable.

Further firmware development could include loudspeaker and room correction. Implementation would equalise loudspeakers and correct for room effects which are located at a substantial distance from the loudspeakers themselves.

List of References

Antoniou, A. 2006. *Digital signal processing: signals, systems, and filters.* pp. 425 - 434, p. 620. McGraw-Hill.

Ashenden, P.J. 2008. *The designer's guide to VHDL.* pp. 1 - 2. Morgan Kaufmann.

Bresch, E. *&* Padgett, W.T. 1999. Tms320c67-based design of a digital audio power amplifier introducing novel feedback strategy. Technical Report, Rose-Hulman Institute of Technology.

Butt, R. 2008. *Introduction to numerical analysis using MATLAB.* pp. 277 - 328. Infinity Science Press.

Candy, B.H. *&* Cox, S.M. 2004. Improved analogue class-d amplifier with carrier symmetry modulation. In *Audio Engineering Society Convention 117.*

Chitode, J. 2009. *Digital signal processing.* pp. 147 - 148. Technical Publications.

Cohen, E., Elber, G. *&* Riesenfeld, R.F. 2001. *Geometric modeling with splines: An introduction.* pp. 291 - 294. A K Peters.

Craven, P. 1993. Toward the 24-bit dac: Novel noise-shaping topologies incorporating correction for the nonlinearity in a pwm output stage. *J. Audio Eng. Soc*, 41(5):291–313.

Crochiere, R. *&* Rabiner, L. 1981. Interpolation and decimation of digital signals; a tutorial review. *Proceedings of the IEEE*, 69(3):300 – 331.

Frerking, M.E. 1994. *Digital signal processing in communication systems.* p. 202. Springer.

Goldberg, J.M. *&* Sandler, M.B. 1991*a*. Noise shaping and pulse-width modulation for an all-digital audio power amplifier. *J. Audio Eng. Soc*, 39(6):449–460.

Goldberg, J.M. *&* Sandler, M.B. 1991*b*. Pseudo-natural pulse width modulation for high accuracy digital-to-analogue conversion. *IEEE Electronics Letters*, 27(16):1491 –1492.

Goyal, M. 2007. *Computer based numerical & statistical techniques.* p. 199, pp. 339 - 356. Jones and Bartlett.

Groenenberg, R., Putzeys, B., van der Hulst, P. *&* Veltman, A. 2006. All amplifiers are analogue, but some amplifiers are more analogue than others. In *Audio Engineering Society Convention 120.*

Hawksford, M. 1989*a*. Chaos, oversampling, and noise shaping in digital-to-analog conversion. *J. Audio Eng. Soc*, 37(12):980–1001.

Hawksford, M. 1989*b*. A tutorial guide to noise shaping and oversampling in adc and dac systems. *Proc. Institute of Acoustics*, 11(7):289 – 302.

Hawksford, M. 2005. Sdm versus pwm power dgital-to-analogue converters (pdac) in high-resolution digital audio applications. In *Audio Engineering Society Convention 118.*

Hayes, M.H. 1999. *Schaum's outline of theory and problems of digital signal processing.* pp. 104 - 106. McGraw-Hill.

Hogenauer, E. 1981. An economical class of digital filters for decimation and interpolation. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 29(2):155 – 162.

Holmes, D.G. *&* Lipo, T.A. 2003. *Pulse width modulation for power converters.* p. 96. Wiley-IEEE.

Jacobs, D. 2006. Digital pulse width modulation for class-d audio amplifiers. Master's thesis, University of Stellebosch.

Kiusalaas, J. 2005. *Numerical methods in engineering with MATLAB.* pp. 103 - 111. Cambridge University Press.

Koeslag, F. 2008. A detailed analysis of the imperfections of pulsewidth modulated waveforms on the output stage of a class d audio amplifier. Ph.D. thesis, University of Stellebosch.

Kozak, M. *&* Kale, I. 2003. *Oversampled delta-sigma modulators: analysis, applications and novel topologies.* p. 2, pp. 194 - 201. Springer.

Logan, S. *&* Hawksford, M. 1994. Linearization of class d output stages for high-performance audio power amplifiers. In *Second international IEE conference on analogue-to-digital and digital-to-analogue conversion, Cambridge university*, pages 136 –141.

Lutovac, M.D., Tosic, D.V. *&* Evans, B.L. 2001. *Filter design for signal processing using MATLAB and Mathematica.* pp. 415 - 416. Prentice Hall.

Lyons, R.G. 2007. *Streamlining digital signal processing: a tricks of the trade guidebook.* p. 174. IEEE Press.

Maloberti, F. 2007. *Data converters.* pp. 312 - 317. Springer.

Meijering, E. 2002. A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proceedings of the IEEE*, 90(3):319 –342.

Mouton, T. *&* Putzeys, B. 2009. Digital control of a pwm switching amplifier with global feedback. In *Audio Engineering Society Conference: 37th International Conference: Class D Audio Amplification.*

Neesgaard, C. *&* Risbo, L. 2006. Pwm amplifier control loops with minimum aliasing distortion. In *Audio Engineering Society Convention 120.*

Nielsen, K. 1998. Audio power amplifier techniques with energy efficient power conversion. Ph.D. thesis, Technical University of Denmark.

Norsworthy, S.R., Schreier, R. *&* Temes, G.C. 1997. *Delta-Sigma data converters: Theory, design, and simulation.* pp. 176 - 180. IEEE Press.

Proakis, J.G. *&* Manolakis, D.G. 1996. *Digital signal processing principles, algorithms, and applications.* pp. 23 - 28. Prentice Hall.

Putzeys, B. 2006. Simple, ultralow distortion digital pulse width modulator. In *Audio Engineering Society Convention 120.*

Risbo, L. 2005. Discrete-time modeling of continuous-time pulse width modulator loops. In *Audio Engineering Society Conference: 27th International Conference: Efficient Audio Power Amplification.*

Schreier, R. *&* Temes, G.C. 2005. *Understanding delta-sigma data converters.* p. 101, pp. 115 - 123. IEEE Press.

Self, D. 2006. *Audio power amplifier design handbook.* pp. 260 - 262, p. 326. Elsevier.

Shannon, C.E. 1949. Communication in the presence of noise. In *Proc. Institute of Radio Engineers*, volume 37, pages 10–21.

Slone, G.R. 2002. *The audiophile's project sourcebook.* pp. 18 - 19. McGraw-Hill.

Stetter, H.J. 2004. *Numerical polynomial algebra.* pp. 163 - 166. Society for Industrial and Applied Math.

Stranneby, D. *&* Walker, W. 2004. *Digital signal processing and applications.* p. 7. Newnes.

Tan, L. 2008. *Digital signal processing.* pp. 7 - 11, pp. 420 - 429. Elsevier.

Tewksbury, S. *&* Hallock, R. 1978. Oversampled, linear predictive and noise-shaping coders of order n $>$ 1. *Circuits and Systems, IEEE Transactions on*, 25(7):436 – 447.

Trefethen, L.N. 2000. *Spectral methods in MATLAB.* pp. 41 - 43. Society for Industrial and Applied Math.

Wescott, T. 2006. *Applied control theory for embedded systems.* pp. 1 - 6, 125 - 128. Newnes.

Appendices

Appendix A

Noise shaper filter coefficients

A.1 Classic noise shaper coefficient design

The transfer function of the noise shaper, where N denotes the order of the filter is as follows:

$$H(z) = [1 - z^{-1}]^N \tag{A.1.1}$$

$$H(z) = \frac{z^N - 1}{z^N} \tag{A.1.2}$$

The denominator of Equation A.1.2 will always equal one. Therefore only the numerator be focused on. The filter coefficients can be found by using the binomial formula, which is shown in equation A.1.3.

$$\begin{aligned}(a+z)^n =& a^n + na^{n-1}z + \frac{n(n-1)}{2!}a^{n-2}z^2 + \frac{n(n-1)(n-2)}{3!}a^{n-3}z^3 \\ &+ \frac{n(n-1)(n-2)(n-3)}{4!}a^{n-4}z^4 + \dots\end{aligned} \tag{A.1.3}$$

1^{st} Order noise shaper:

$$\begin{aligned}NTF(z) =& (-1)^1 + (1)(-1)^0 z^{-1} \\ =& -1 + z^{-1}\end{aligned} \tag{A.1.4}$$

Coefficients = [-1 1]

2^{nd} Order noise shaper:

$$\begin{aligned} NTF(z) =&(-1)^2 + 2(-1)^1 z^{-1} + \frac{2}{2!}(-1)^0 z^{-2} \\ =&1 - 2z^{-1} + z^{-2} \end{aligned} \tag{A.1.5}$$

Coefficients = [1 -2 1]

3^{rd} Order noise shaper:

$$\begin{aligned} NTF(z) =&(-1)^3 + 3(-1)^2 z^{-1} + \frac{6}{2}(-1)^1 z^{-2} + \frac{3(2)(1)}{6}(-1)^0 z^{-3} \\ =&-1z^0 + 3z^{-1} - 3z^{-2} + 1z^{-3} \end{aligned} \tag{A.1.6}$$

Coefficients = [-1 3 -3 1]

4^{th} Order noise shaper:

$$\begin{aligned} NTF(z) =&(-1)^4 + 4(-1)^4 z^{-1} + 6(-1)^2 z^{-2} + \frac{24}{6}(-1) z^{-3} + (-1)^0 z^{-4} \\ =&1 - 4z^{-1} + 6z^{-2} - 4z^{-3} + z^{-4} \end{aligned} \tag{A.1.7}$$

Coefficients = [1 -4 6 -4 1]

5^{th} Order noise shaper:

$$\begin{aligned} NTF(z) =&(-1)^5 + 5(-1)^4 z^{-1} + 10(-1)^3 z^{-2} + 10(-1)^2 z^{-3} + 5(-1)^1 z^{-4} + 1(-1)^0 z^{-5} \\ =&-1 + 5z^{1-} - 10z^{-2} + 10z^{-3} - 5z^{-4} + z^{-5} \end{aligned} \tag{A.1.8}$$

Coefficients = [-1 5 -10 10 -5 1]

Appendix B

VHDL source code

B.1 Main program source code

Listing B.1: Main program

```
-- Design Unit : Main program file of project
-- File name   : main.vhd
-- Description : Central file for calling subsections and directing data
--
-- Author      : Jason Quibell
--               Centre for Instrumentation Research
--               Cape Peninsula University of Technology
-- Revision    : Version 1.9 07/01/2011

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

Library ieee_proposed;
use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;

entity dig_amp is

port (
  CLK         : in std_logic;     --24.576 MHz from Oscillator
  pwm_out     : out std_logic;
  CS          : out std_logic;    -- SPI Chip Select
  CDIN        : out std_logic;    -- SPI Data In
  SCLKo       : out std_logic;    -- SPI Serial Clock
  SPI_reset   : out std_logic;    -- SPI reset

  SRC_MCLK  : out std_logic;      -- SRC master clock
  SRC_SDOUT : in std_logic;    -- SRC Serial Data Out
  SRC_LRCKA : out std_logic;   -- SRC Left/Right Clock
  SRC_BCKA  : out std_logic    -- SRC Bit Clock
```

```

    );
end entity dig_amp;

architecture dig_amp_a of dig_amp is

component SPI_M
  port (
       clk        : in  std_logic;
       SCLK       : out std_logic;
       SS         : out std_logic;
       MOSI       : out std_logic
       );
  end component;

component dataRX_SRC
  port (
    clock      : in std_logic;
    doneRX     : out std_logic;  --Activate "Receive Six values"
    LRclk      : in std_logic;   --Left/Right Clock
    Bclk       : in std_logic;   --Bit Clock
    AudioData  : in std_logic;   --Audio data bit stream in
    AudioWord  : out signed(23 downto 0) --Audio Data Word
       );
  end component;

component RX_six
  port (
    clock      : in std_logic;
    doneRX     : in std_logic;   -- check when finished RX
    audio_word: in signed(23 downto 0);  --Audio data in
    run_coeff  : out std_logic;  --Activate "Coefficient calc."
    audio0     : out signed(23 downto 0);    -- Six audio values
    audio1     : out signed(23 downto 0);
    audio2     : out signed(23 downto 0);
    audio3     : out signed(23 downto 0);
    audio4     : out signed(23 downto 0);
    audio5     : out signed(23 downto 0)
    );
end component;

component PolyCoeff
  port (
    clock      : in std_logic;
    run_coeff  : in std_logic;  --Activate "Coefficient calc."
    coeff_done: out std_logic; --Activate "Coeff. done"

    coeff0     : out signed(23 downto 0); --Polynomial Coefficients
    coeff1     : out signed(23 downto 0);
    coeff2     : out signed(23 downto 0);
    coeff3     : out signed(23 downto 0);
    coeff4     : out signed(23 downto 0);
```

```
      coeff5      : out signed(23 downto 0);

      audio0      : in signed(23 downto 0);      -- Six audio values
      audio1      : in signed(23 downto 0);
      audio2      : in signed(23 downto 0);
      audio3      : in signed(23 downto 0);
      audio4      : in signed(23 downto 0);
      audio5      : in signed(23 downto 0)
      );
      END COMPONENT;


COMPONENT PolyCalc
   PORT(
      clock         : in std_logic;
      coeff_done    : in std_logic;
         check      : out std_logic;
         counter    : out sfixed(23 downto -6);

      coeff0        : in signed(23 downto 0);
      coeff1        : in signed(23 downto 0);
      coeff2        : in signed(23 downto 0);
      coeff3        : in signed(23 downto 0);
      coeff4        : in signed(23 downto 0);
      coeff5        : in signed(23 downto 0);
      upsampled     : out sfixed(0 downto -23)
      );
      END COMPONENT;

COMPONENT COMPENSATOR
   PORT(
      clk           : in std_logic;
      filtIn        : in sfixed(1 downto -23);
      filtOut       : out sfixed(1 downto -23)
      );
      END COMPONENT;

COMPONENT PWM
   PORT(
      clock         : in std_logic;
      upsampled     : in sfixed(0 downto -23);
      filtOut       : in sfixed(1 downto -23);
      filtIn      : out sfixed(1 downto -23);
        pwmFB       : out sfixed(1 downto -23);
      pwmOut        : out std_logic
      );
      END COMPONENT;

component pll
   PORT
   (
      areset        : IN STD_LOGIC  := '0';
      inclk0        : IN STD_LOGIC  := '0';
      c0            : OUT STD_LOGIC ;
      c1            : OUT STD_LOGIC ;
      c2            : OUT STD_LOGIC ;
      locked        : OUT STD_LOGIC
   );
end component;




 -- component ports
   signal reset       : std_logic;  -- SPI reset
   signal SCLK        : std_logic;  -- Slave Clock , SPI
   signal SS          : std_logic;  -- Slave Select / Chip Select
   signal MOSI        : std_logic;  -- Master out Slave in,  SPI data
   signal pllCLK      : std_logic;  -- 49.152 MHz clock from PLL
   signal bCLK        : std_logic;  -- 12.288 MHz -Bit Clock
   signal wCLK        : std_logic;  -- 192 kHz   -Word Clock
   signal doneRX      : std_logic;  -- Receiving audio complete

   signal pllreset    : std_logic;
   signal nlocked     : std_logic;
   signal run_coeff   : std_logic;
   signal coeff_done: std_logic;

   signal check       : std_logic;
   signal counter     : sfixed(23 downto -6);


   signal AudioLeft   : signed(23 downto 0); --Left channel audio
   signal audio0      : signed(23 downto 0);
   signal audio1      : signed(23 downto 0);
   signal audio2      : signed(23 downto 0);
   signal audio3      : signed(23 downto 0);
   signal audio4      : signed(23 downto 0);
   signal audio5      : signed(23 downto 0);

   signal coeff0      : signed(23 downto 0);
   signal coeff1      : signed(23 downto 0);
   signal coeff2      : signed(23 downto 0);
   signal coeff3      : signed(23 downto 0);
   signal coeff4     : signed(23 downto 0);
   signal coeff5     : signed(23 downto 0);

   signal filtIn      : sfixed(1 downto -23);
   signal filtOut     : sfixed(1 downto -23);

   signal upsampled   : sfixed(0 downto -23);
   signal pwmout      : std_logic;

begin
```

```

-- component instantiation
 DUT1: SPI_M
   port map (
     clk        => CLK,   --24.576 MHz
     SCLK       => SCLK,
     SS         => SS,
     MOSI       => MOSI
     );

 DUT2: dataRX_SRC
 port map (
     clock      => CLK, --24.576 MHz
     doneRX     => doneRX,
     LRclk      => wCLK,
     Bclk       => bCLK,
     AudioWord  => AudioLeft,
     AudioData  => SRC_SDOUT
     );

 DUT3: RX_six
 port map (
   clock        => CLK, --24.576 MHz
   doneRX       => doneRX,
   audio_word   => AudioLeft,
   run_coeff    => run_coeff,
   audio0       => audio0,
   audio1       => audio1,
   audio2       => audio2,
   audio3       => audio3,
   audio4       => audio4,
   audio5       => audio5
   );

 DUT4: PolyCoeff
 port map (
   clock        => pllCLK,   --49.152 MHz
   run_coeff    => run_coeff,
   coeff_done   => coeff_done,
   coeff0       => coeff0,
   coeff1       => coeff1,
   coeff2       => coeff2,
   coeff3       => coeff3,
   coeff4       => coeff4,
   coeff5       => coeff5,
   audio0       => audio0,
   audio1       => audio1,
   audio2       => audio2,
   audio3       => audio3,
   audio4       => audio4,
   audio5       => audio5
   );
```

```vhdl

 DUT5: PolyCalc
 port map (
     clock         => pllCLK,   --49.152 MHz
     check         => check,
     counter       => counter,
     coeff_done    => coeff_done,
     coeff0        => coeff0,
     coeff1        => coeff1,
     coeff2        => coeff2,
     coeff3        => coeff3,
     coeff4        => coeff4,
     coeff5        => coeff5,
     upsampled     => upsampled
     );

DUT6: COMPENSATOR
 PORT MAP(
     clk           => pllCLK,   --49.152 MHz
     filtIn        => filtIn,   --Compensated for ripple
     filtOut       => filtOut
     );

DUT7: PWM
 PORT MAP(
     clock         => pllCLK,   --49.152 MHz
     filtOut       => filtOut,   --Upsampled Filtered Audio
     filtIn        => filtIn,
     upsampled=> upsampled,
     pwmOut        => pwmout,
     );


  u_pll: pll                    --PLL running
    PORT MAP (
                 inclk0    => CLK,       --24.576 MHz - External Osc.
                 c0        => pllCLK,   --49.152 MHz
                 c1        => bCLK,      --12.288 MHz - Bit Clock
                 c2        => wCLK,      --   192 kHz - Word Clock
                 locked    => nlocked,
                 areset    => pllreset
                );

process(CLK, pllCLK, wCLK, bCLK, SCLK)
 begin

     SPI_reset          <= '1';
    SRC_MCLK            <= CLK;   --24.576 MHz
    CS                  <= SS;    --SPI chip select
    CDIN                <= MOSI;  --SPI data in
    SCLKo               <= SCLK;  --SPI data clock
```

```
        SRC_LRCKA          <= wCLK; -- SRC Left/Right Clock   192 kHz
        SRC_BCKA           <= bCLK; -- SRC Bit Clock  12.288 MHz

        PWM_out            <= pwmout;

    end process;

end architecture dig_amp_a;
```

B.2 SPI source code

Listing B.2: SPI

```
-- Design Unit : SPI Master
-- File name   : SPI_M.vhd
-- Description : Use the FPGA as an SPI master to configure the
   SRC4392 device. Specific data words are addressed and sent to
   the SRC4392 device. Once complete, the FPGA stops sending any
   further data via SPI.
--
-- Author      : Jason Quibell
--               Centre for Instrumentation Research
--               Cape Peninsula University of Technology
-- Revision    : Version 1.1 05/05/2009

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SPI_M is
  port (
    clk      : in  std_logic;
    SCLK     : out std_logic;
    SS       : out std_logic;
    MOSI     : out  std_logic
    );
end SPI_M;

architecture a of SPI_M is
    type state_type is (idle, loadData, delay1, txBit,
       CheckFinished);
    signal state   : state_type := idle;

    signal DataToTx: std_logic_vector(23 downto 0);
    signal StartTX : std_logic := '1';
    signal counter : unsigned(3 downto 0);

begin
  process(clk,counter)

    variable index    : integer := 0;
    variable dataLen  : integer := 23;   -- length of data to TX
    variable MOSI_v   : std_logic;  --Master Out Slave In

  begin

    if(counter = 0) then
    DataToTx <= "110011000000000010000000"; -- register 01
  elsif(counter = 1) then
    DataToTx <= "001011000000000011000000"; --register 03
  elsif(counter = 2) then
    DataToTx <= "010100000000000010110000"; --register 0D
  elsif(counter = 3) then
    DataToTx <= "100100000000000001110000"; --register 0E
  elsif(counter = 4) then
    DataToTx <= "010001000000000011110000"; --register 0F
  elsif(counter = 5) then
    DataToTx <= "010000100000000010110100"; -- register 2D
  elsif(counter = 6) then
    DataToTx <= "000000110000000011110100"; -- register 2F
  elsif(counter = 7) then
    DataToTx <= "001000000000000000001100"; -- register 30
  elsif(counter = 8) then
    DataToTx <= "000000000000000000000000"; --Disable SPI
    end if;

        if(clk'event and clk = '1') then
         case state is
           when idle =>
             SCLK <= '0';
             SS <= '1';                          -- stop SPI
             MOSI_v := '0';
             if(StartTx = '1') then
               state <= loadData;
             else
               state <= idle;
               index := 0;
              end if;

           when loadData =>
             SS <= '0';                          -- start SPI
             SCLK <= '0';
             MOSI_v := DataToTx(index);          -- TX data to slave
             state <= delay1;

           when delay1 =>
            state <= txBit;

           when txBit =>
             SCLK <= '1';
             state <= CheckFinished;

           when checkFinished =>
             if(index = dataLen) then
               if counter = 8 then
                 SS <= '1';
               else
                 index := 0;
                 counter <= counter + 1;
                 state <= idle;
               end if;
             else
                 state <= loadData;
                 index := index + 1;
           end if;

          when others => null;
        end case;

      end if;
    MOSI <= MOSI_v;
  end process;
end a;
```

B.3 Receive audio data source code

Listing B.3: Receive audio data word

```
-- Design Unit : Receive audio data from SRC device
-- File name   : dataRX_SRC.vhd
-- Description : Receive audio data being transmitted by SRC4392 device and pass it to the next block which collects six audio data words to be used for a fifth order polynomial calculation.
--
-- Author      : Jason Quibell
--               Centre for Instrumentation Research
--               Cape Peninsula University of Technology
-- Revision    : Version 1.2 17/05/2009

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dataRX_SRC is
  port (
    clock     : in std_logic;
    LRclk     : in std_logic;
    Bclk      : in std_logic;
    AudioData: in std_logic;
    AudioWord: out signed(23 downto 0);
    doneRX    : out std_logic
      );
end dataRX_SRC;

architecture a of dataRX_SRC is
  type state_type is (go, stop);
  signal state : state_type := stop;


   signal DataRX        : signed(15 downto 0);     --Actual 16bit received data
   signal    Audio_L    : signed(23 downto 0);     --16bit + 8bit padding (24bit)

begin
  process(LRclk,Bclk)
  begin
    if LRclk = '1' then
      if Bclk'event and Bclk ='0' then
          DataRX(15 downto 1) <= DataRX(14 downto 0);
          DataRX(0) <= AudioData;
          doneRX    <= '0';
      end if;
    else

      Audio_L(7 downto 0)   <= "00000000";
      Audio_L(23 downto 8)  <= DataRx;
```

```
        AudioWord <= Audio_L;
        doneRX    <= '1';
    end if;
  end process;


end a;
```

Listing B.4: Store six audio words

```vhdl
-- Design Unit : Store six audio words
-- File name   : dataRX_SRC.vhd
-- Description : Received audio data is stored until six values
    have been accumulated. They are then passed to the polynomial
    coefficient calculator
--
-- Author      : Jason Quibell
--               Centre for Instrumentation Research
--               Cape Peninsula University of Technology
-- Revision    : Version 1 13/05/2009

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity RX_six is
  port (
    clock       : in std_logic;
    doneRX      : in std_logic;    -- check when done RX data
    audio_word  : in signed(23 downto 0);  --Audio data in
    run_coeff   : out std_logic;  --Activate "Coefficient calc"
    audio0      : out signed(23 downto 0); -- Six audio values
    audio1      : out signed(23 downto 0);
    audio2      : out signed(23 downto 0);
    audio3      : out signed(23 downto 0);
    audio4      : out signed(23 downto 0);
    audio5      : out signed(23 downto 0)
     );
end RX_six;

architecture a of RX_six is

  type state_type is (idle, acceptData, newValue, dataOut, delay,
      delay2);
  signal state : state_type := idle;

  type audio_arr is array (INTEGER range <>) of signed(23 downto
      0);
  signal audio_array : audio_arr(0 to 5); -- a 6 x 24bit array


begin
  process(clock, audio_array)


  begin
  if(clock'event and clock = '1') then
        case state is

          when idle =>
            run_coeff <= '0';   --Deactivate Coefficient Calc.

            if doneRX = '1' then
              state <= acceptData;
            else
              state <= idle;
            end if;

          when acceptData =>
          audio_array(0) <= audio_array(1);
          audio_array(1) <= audio_array(2);
          audio_array(2) <= audio_array(3);
          audio_array(3) <= audio_array(4);
          audio_array(4) <= audio_array(5);
          state          <= newValue;

          when newValue =>
        audio_array(5)   <= audio_word; --Introduce new data
        state            <= dataOut;

          when dataOut =>
        run_coeff <= '1'; --Activate Coefficient Calculator
        audio0 <= audio_array(0); --6 values for coefficient calc.
        audio1 <= audio_array(1);
        audio2 <= audio_array(2);
        audio3 <= audio_array(3);
        audio4 <= audio_array(4);
        audio5 <= audio_array(5);
        state  <= delay;

      when delay =>
        if doneRX = '0' then
          state <= idle;
        else
          state <= delay2;
        end if;

      when delay2 =>
        run_coeff <= '0';
        state <= delay;


          when others => null;
        end case;
  end if;
   end process;
end a;
```

B.4 Interpolation source code

Listing B.5: Calculate polynomial coefficients

```vhdl
-- Design Unit : Calculate polynomial coefficients
-- File name   : PolyCoeff.vhd
-- Description : Six audio data values are received and used to create fifth order polynomial coefficients. The coefficients are then passed to the next block which then generates new data , interpolation.
--
-- Author      : Jason Quibell
--               Centre for Instrumentation Research
--               Cape Peninsula University of Technology
-- Revision    : Version 1.4 11/06/2009

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

Library ieee_proposed;
use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;

entity PolyCoeff is
  port (
      clock      : in std_logic;
    run_coeff    : in std_logic; --Activate "Coefficient calculator"
    coeff_done   : out std_logic; --Activate "Coefficient done"

    coeff0     : out signed(23 downto 0);  --Six coefficients
    coeff1     : out signed(23 downto 0);
    coeff2     : out signed(23 downto 0);
    coeff3     : out signed(23 downto 0);
    coeff4     : out signed(23 downto 0);
    coeff5     : out signed(23 downto 0);

    audio0     : in signed(23 downto 0);    -- Six audio data values
    audio1     : in signed(23 downto 0);
    audio2     : in signed(23 downto 0);
    audio3     : in signed(23 downto 0);
    audio4     : in signed(23 downto 0);
    audio5     : in signed(23 downto 0)
      );
end PolyCoeff;

architecture a of PolyCoeff is

  type state_type is (idle, calcCoeff1, calcCoeff2, calcCoeff3, calcCoeff4, calcCoeff5, coeffDone, delay);
  signal state : state_type := idle;

  type coeff_arr is array (INTEGER range <>) of signed(23 downto
      0);
  signal coeff_0 : coeff_arr(0 to 5); -- a 6 x 24bit array
  signal coeff_1 : coeff_arr(0 to 4); -- a 5 x 24bit array
  signal coeff_2 : coeff_arr(0 to 3); -- a 4 x 24bit array
  signal coeff_3 : coeff_arr(0 to 2); -- a 3 x 24bit array
  signal coeff_4 : coeff_arr(0 to 1); -- a 2 x 24bit array
  signal coeff_5 : coeff_arr(0 to 0); -- a 1 x 24bit array




begin
  process(clock, coeff_0, coeff_1, coeff_2, coeff_3, coeff_4,
      coeff_5)

  variable index : integer := 0;

  begin
  if(clock'event and clock = '1') then
        case state is

          when idle =>
          coeff_done <= '0';
            if run_coeff = '1' then
              coeff_0(0) <= audio0;
              coeff_0(1) <= audio1;
              coeff_0(2) <= audio2;
              coeff_0(3) <= audio3;
              coeff_0(4) <= audio4;
              coeff_0(5) <= audio5;
              state <= calcCoeff1;
            else
              state <= idle;
            end if;

      when calcCoeff1 =>
        coeff_1(0) <= coeff_0(1) - coeff_0(0);
        coeff_1(1) <= coeff_0(2) - coeff_0(1);
        coeff_1(2) <= coeff_0(3) - coeff_0(2);
        coeff_1(3) <= coeff_0(4) - coeff_0(3);
        coeff_1(4) <= coeff_0(5) - coeff_0(4);
      state <= calcCoeff2;

      when calcCoeff2 =>
        coeff_2(0) <= (coeff_1(1) - coeff_1(0))/2;
        coeff_2(1) <= (coeff_1(2) - coeff_1(1))/2;
        coeff_2(2) <= (coeff_1(3) - coeff_1(2))/2;
        coeff_2(3) <= (coeff_1(4) - coeff_1(3))/2;
      state <= calcCoeff3;
      when calcCoeff3 =>
        coeff_3(0) <= (coeff_2(1) - coeff_2(0))/3;
        coeff_3(1) <= (coeff_2(2) - coeff_2(1))/3;
        coeff_3(2) <= (coeff_2(3) - coeff_2(2))/3;
      state <= calcCoeff4;


      when calcCoeff4 =>
        coeff_4(0) <= (coeff_3(1) - coeff_3(0))/4;
        coeff_4(1) <= (coeff_3(2) - coeff_3(1))/4;
      state <= calcCoeff5;


      when calcCoeff5 =>
        coeff_5(0)  <= (coeff_4(1) - coeff_4(0))/5;
        coeff0    <=  coeff_0(0);
        coeff1    <=  coeff_1(0);
        coeff2    <=  coeff_2(0);
        coeff3    <=  coeff_3(0);
        coeff4    <=  coeff_4(0);
        coeff5    <=  coeff_5(0);
        coeff_done  <=  '1';
      state <= coeffDone;

      when coeffDone =>
        coeff_done  <=  '1';
        state       <= delay;

      when delay  =>
        state     <=  idle;

          when others => null;
        end case;
  end if;
   end process;
end a;
```

Listing B.6: Calculate new audio data

```
-- Design Unit : Generate new data from polynomial coefficients
-- File name   : PolyCalc.vhd
-- Description : Receive polynomial coefficients and generate new data, thereby increases the number of samples available, which then makes it possible to increase the sample rate at which the system operates. 64 new samples are generated for every single sample which is received by the FPGA. Pre-calculated values were stored in ROM.
--
-- Author      : Jason Quibell
--               Centre for Instrumentation Research
--               Cape Peninsula University of Technology
-- Revision    : Version 1.6 17/08/2009

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

Library ieee_proposed;
use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;

entity PolyCalc is
  port (
    clock     : in std_logic;
    coeff_done: in std_logic;

    coeff0    : in signed(23 downto 0);
    coeff1    : in signed(23 downto 0);
    coeff2    : in signed(23 downto 0);
    coeff3    : in signed(23 downto 0);
    coeff4    : in signed(23 downto 0);
    coeff5    : in signed(23 downto 0);

    check     : out std_logic;
    counter   : out sfixed(23 downto -6);

    upsampled : out sfixed(0 downto -23)
      );
end PolyCalc;

architecture a of PolyCalc is

  type state_type is (idle, calc, delay, delay2, xCheck);
  signal state : state_type := idle;

  type Data_arr is array (INTEGER range <>) of sfixed(23 downto -6);
  signal a     : Data_arr(0 to 5); -- Coefficients in 6x24bit array resized to 23 downto -6
  signal x        : sfixed(23 downto -6) := "
      000000000000000000000000000000";
  signal p        : sfixed(23 downto -6);
  signal p_tmp    : sfixed(23 downto -6);

  signal six      : sfixed(23 downto -6);
  signal add      : sfixed(23 downto -6);
  signal zero     : sfixed(23 downto -6);

  signal address_sig   :UNSIGNED (5 DOWNTO 0) := "000000";
  signal address_tmp   :UNSIGNED (5 DOWNTO 0) := "000000";
  signal q_sig1        :sfixed(23 downto -6);
  signal q_sig2        :sfixed(23 downto -6);
  signal q_sig3        :sfixed(23 downto -6);
  signal q_sig4        :sfixed(23 downto -6);
  signal q_sig5        :sfixed(23 downto -6);
  signal q_sig1_tmp    :sfixed(23 downto -6);
  signal q_sig2_tmp    :sfixed(23 downto -6);
  signal q_sig3_tmp    :sfixed(23 downto -6);
  signal q_sig4_tmp    :sfixed(23 downto -6);
  signal q_sig5_tmp    :sfixed(23 downto -6);



component ROM
  PORT
  (
    address         : IN UNSIGNED (5 DOWNTO 0);
    clock           : IN STD_LOGIC ;
    q               : OUT sfixed(23 downto -6)
  );
end component;

component ROM2
  PORT
  (
    address         : IN UNSIGNED (5 DOWNTO 0);
    clock           : IN STD_LOGIC ;
    q               : OUT sfixed(23 downto -6)
  );
end component;

component ROM3
  PORT
  (
    address         : IN UNSIGNED (5 DOWNTO 0);
    clock           : IN STD_LOGIC ;
    q               : OUT sfixed(23 downto -6)
  );
end component;
```

```
component ROM4
  PORT
   (
     address    : IN UNSIGNED (5 DOWNTO 0);
     clock      : IN STD_LOGIC ;
     q          : OUT sfixed(23 downto -6)
   );
end component;

component ROM5
  PORT
   (
     address    : IN UNSIGNED (5 DOWNTO 0);
     clock      : IN STD_LOGIC ;
     q          : OUT sfixed(23 downto -6)
   );
end component;


begin
--  Pre-calculated values
ROM_inst  : ROM PORT MAP (
     address   => address_sig,
     clock     => clock,
     q         => q_sig1_tmp
   );

ROM_inst2  : ROM2 PORT MAP (
     address   => address_sig,
     clock     => clock,
     q         => q_sig2_tmp
   );
ROM_inst3  : ROM3 PORT MAP (
     address   => address_sig,
     clock     => clock,
     q         => q_sig3_tmp
   );

ROM_inst4  : ROM4 PORT MAP (
     address   => address_sig,
     clock     => clock,
     q         => q_sig4_tmp
   );

ROM_inst5  : ROM5 PORT MAP (
     address   => address_sig,
     clock     => clock,
     q         => q_sig5_tmp
   );


   process(clock)
```

```

variable index : integer := 0;

begin

    six    <= to_sfixed(4.96875,six); --Upper limit
    add    <= to_sfixed(0.015625,add);--Step size(64x OSR)
    zero   <= to_sfixed(4,zero);      --Lower limit

if(clock'event and clock = '1') then
      case state is

        when idle =>
    if coeff_done = '1' then

      a(0)(23 downto 0)  <= to_sfixed(coeff0,23,0);
      a(1)(23 downto 0)  <= to_sfixed(coeff1,23,0);
      a(2)(23 downto 0)  <= to_sfixed(coeff2,23,0);
      a(3)(23 downto 0)  <= to_sfixed(coeff3,23,0);
      a(4)(23 downto 0)  <= to_sfixed(coeff4,23,0);
      a(5)(23 downto 0)  <= to_sfixed(coeff5,23,0);

      state <= calc;
    else
      check <= '0';
      state <= idle;
    end if;

          when calc =>

    p_tmp   <= resize(a(0) + q_sig1*a(1) + q_sig2*a(2) + q_sig3*
       a(3) + q_sig4*a(4) + q_sig5*a(5),p_tmp'high,p_tmp'low);
    address_tmp <= address_tmp + 1;
    check   <= '1';
    state   <= delay;

    when delay =>

    state <= delay2;

    when delay2 =>

    state <= xCheck;

    when xCheck =>
    upsampled <= p_tmp(23 downto 0);

    if x = six then
      x           <= zero;
      address_tmp <= "000000";
      state       <=  idle;
    else
```

```
        x       <= resize(x + add,x'high,x'low);
        counter <= x;
        check   <= '0';
        state   <= calc;
      end if;

          when others => null;
        end case;
  end if;
   end process;

  process(clock)
    begin
    if(clock'event and clock = '1') then
      address_sig <= address_tmp;
      q_sig1      <= q_sig1_tmp;
      q_sig2      <= q_sig2_tmp;
      q_sig3      <= q_sig3_tmp;
      q_sig4      <= q_sig4_tmp;
      q_sig5      <= q_sig5_tmp;
    end if;
  end process;
end a;
```

B.5 Noise shaper source code

Listing B.7: 3^{rd} Order noise shaping loop filter

```
-- Design Unit : 3rd Order Loop Filter
-- File name   : COMPENSATOR3rdOrder.vhd
-- Description : Filter upsampled audio with a 3rd order cascaded
    integrators with feedforward summation and resonator feedback
    filter. The filter coefficients were calculated offline.
--
-- Author      : Jason Quibell
--               Centre for Instrumentation Research
--               Cape Peninsula University of Technology
-- Revision    : Version 1.4 02/04/2011

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

Library ieee_proposed;
use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;


--------------------------------------------------------------------------
entity COMPENSATOR is
  port (
    clk      : in std_logic;
    filtIn    : in sfixed(1 downto -23);
    filtOut   : out sfixed(1 downto -23)
      );
end COMPENSATOR;

--------------------------------------------------------------------------
--Module Architecture: compensator
--------------------------------------------------------------------------
ARCHITECTURE rtl OF COMPENSATOR IS

  -- Coefficients



 --3rd Order coefficients

 -- 12.288 MHz
  CONSTANT a1          : sfixed(0 downto -34) := "
       000001000000110001101000100101101111";
  CONSTANT a2          : sfixed(0 downto -39) := "
       0000000001001001001111110011111001000001";
  CONSTANT a3          : sfixed(0 downto -41) := "
       000000000000000100111000011110110011100010";

  CONSTANT g1           : sfixed(0 downto -39) := "
      0000000000000000011110110101111010111010";

  -- Signals
  SIGNAL Y1                       : sfixed(8 DOWNTO -31):= "
      0000000000000000000000000000000000000000";
  SIGNAL Y2                       : sfixed(14 DOWNTO -31):= "
      0000000000000000000000000000000000000000000000";
  SIGNAL Y3                       : sfixed(22 DOWNTO -31):= "
      000000000000000000000000000000000000000000000000000000";

  --Delay variables
  SIGNAL Y1_delay                 : sfixed(8 DOWNTO -31):= "
      0000000000000000000000000000000000000000";
  SIGNAL Y2_delay                 : sfixed(14 DOWNTO -31):= "
      0000000000000000000000000000000000000000000000";
  SIGNAL Y3_delay                 : sfixed(22 DOWNTO -31):= "
      000000000000000000000000000000000000000000000000000000";

  --Final filtered output
  SIGNAL filtered            : sfixed(1 downto -23):= "
      0000000000000000000000000";

  --Temporary calculation storage
  SIGNAL tmp_one          : sfixed(1 downto -23):= "
      0000000000000000000000000";
   signal tmp_two          : sfixed(1 downto -23):= "
      0000000000000000000000000";
   signal tmp_three           : sfixed(1 downto -23):= "
      0000000000000000000000000";

   signal Y3_feedback         : sfixed(1 downto -31):= "
      000000000000000000000000000000000";


   type state_type is (start, delay1, delay2, stop);
   signal state : state_type := start;

begin

   delay_process_section : process (clk)
   begin
   if clk'event AND clk = '1' THEN

 case state is

   when start =>

        filtOut <= resize(tmp_one + tmp_two + tmp_three,filtered'
            high,filtered'low);
      Y2 <= resize(Y1_delay  + Y2_delay - Y3_feedback,Y2'high,Y2'low);

  state <= delay1;

   when delay1 =>

      Y1 <= resize(filtIn + Y1_delay,Y1'high,Y1'low);
      Y3 <= resize(Y2_delay + Y3_delay,Y3'high,Y3'low);
      Y3_feedback <= resize(g1*Y3_delay,Y3_feedback'high,Y3_feedback'low);

    state <= delay2;

   when delay2 =>
      tmp_one <= resize(Y1*a1,tmp_one'high,tmp_one'low);
      tmp_two <= resize(Y2*a2,tmp_two'high,tmp_two'low);
      tmp_three <= resize(Y3*a3,tmp_three'high,tmp_three'low);

  state <= stop;

   when stop =>

      Y1_delay <= Y1;
      Y2_delay <= Y2;
      Y3_delay <= Y3;

   state <= start;
      when others => null;

    end case;
    end if;

  end process delay_process_section;

end rtl;
```

Listing B.8: 5^{th} Order noise shaping loop filter

```
-- Design Unit : 5th Order Loop Filter
-- File name   : COMPENSATOR5thOrder.vhd
-- Description : Filter upsampled audio with a 5th order cascaded
    integrators with feedforward summation and resonator feedback
    filter. The filter coefficients were calculated offline.
--
-- Author      : Jason Quibell
--               Centre for Instrumentation Research
--               Cape Peninsula University of Technology
-- Revision    : Version 1.4 02/04/2011



library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

Library ieee_proposed;
use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;


-------------------------------------------------------------------------------
entity COMPENSATOR is
  port (
     clk       : in std_logic;
     filtIn     : in sfixed(1 downto -23);
     filtOut    : out sfixed(1 downto -23)
       );
end COMPENSATOR;

--------------------------------------------------------------------------------
--Module Architecture: compensator
--------------------------------------------------------------------------------
ARCHITECTURE rtl OF COMPENSATOR IS

 -- 5th Order coefficients
  -- 12.288 MHz
  CONSTANT a1          : sfixed(0 downto -34) := "
      00000101100001001000111000110011010";
  CONSTANT a2          : sfixed(0 downto -39) := "
      0000000001001001010101111100000000010101";
  CONSTANT a3          : sfixed(0 downto -41) := "
      000000000000001101101111001000100000101100";
  CONSTANT a4          : sfixed(0 downto -44) := "
      000000000000000000001011010110111010010000000";
  CONSTANT a5          : sfixed(0 downto -49) := "
      00000000000000000000000000000111111111110011110000";
  CONSTANT g1          : sfixed(0 downto -39) := "
      0000000000000101001011101110111000000000";
  CONSTANT g2            : sfixed(0 downto -39) := "
      0000000000000000101111001101111010000010";

  -- Signals
  SIGNAL Y1                    : sfixed(8 DOWNTO -31):= "
      0000000000000000000000000000000000000000";
  SIGNAL Y2                    : sfixed(14 DOWNTO -31):= "
      0000000000000000000000000000000000000000000000";
  SIGNAL Y3                    : sfixed(22 DOWNTO -31):= "
      000000000000000000000000000000000000000000000000000000";
  SIGNAL Y4                    : sfixed(30 DOWNTO -31):= "
      000000000000000000000000000000000000000000000000000000
00000000";
  SIGNAL Y5                    : sfixed(39 DOWNTO -31):= "
      000000000000000000000000000000000000000000000000000000
00000000000000000";


  --Delay variables
  SIGNAL Y1_delay              : sfixed(8 DOWNTO -31):= "
      0000000000000000000000000000000000000000";
  SIGNAL Y2_delay              : sfixed(14 DOWNTO -31):= "
      0000000000000000000000000000000000000000000000";
  SIGNAL Y3_delay              : sfixed(22 DOWNTO -31):= "
      000000000000000000000000000000000000000000000000000000";
  SIGNAL Y4_delay              : sfixed(30 DOWNTO -31):= "
      000000000000000000000000000000000000000000000000000000
00000000";
  SIGNAL Y5_delay                    : sfixed(39 DOWNTO -31):= "
      000000000000000000000000000000000000000000000000000000
00000000000000000";

  --Final filtered output
  SIGNAL filtered          : sfixed(1 downto -23):= "
      0000000000000000000000000";

  --Temporary calculation storage
  SIGNAL tmp_one         : sfixed(1 downto -23):= "
      0000000000000000000000000";
   signal tmp_two        : sfixed(1 downto -23):= "
      0000000000000000000000000";
   signal tmp_three        : sfixed(1 downto -23):= "
      0000000000000000000000000";
   signal tmp_four       : sfixed(1 downto -23):= "
      0000000000000000000000000";
   signal tmp_five       : sfixed(1 downto -23):= "
      0000000000000000000000000";

   signal Y3_feedback      : sfixed(1 downto -31):= "
      000000000000000000000000000000000";
   signal Y5_feedback      : sfixed(1 downto -31):= "
      000000000000000000000000000000000";
   type state_type is (start, delay1, delay2, stop);
   signal state : state_type := start;

 begin

   delay_process_section : process (clk)
   begin
   if clk'event AND clk = '1' THEN

   case state is

    when start =>

       filtOut <= resize(tmp_one + tmp_two + tmp_three + tmp_four +
            tmp_five,filtered'high,filtered'low);
       Y2 <= resize(Y1_delay + Y2_delay - Y3_feedback,Y2'high,Y2'
           low);
       Y4 <= resize(Y3_delay + Y4_delay - Y5_feedback,Y4'high,Y4'
           low);

     state <= delay1;

    when delay1 =>

       Y1 <= resize(filtIn + Y1_delay,Y1'high,Y1'low);
       Y3 <= resize(Y2_delay + Y3_delay,Y3'high,Y3'low);
       Y3_feedback <= resize(g1*Y3_delay,Y3_feedback'high,
           Y3_feedback'low);
       Y5 <= resize(Y4_delay + Y5_delay,Y5'high,Y5'low);
       Y5_feedback <= resize(g2*Y5_delay,Y5_feedback'high,
           Y5_feedback'low);

     state <= delay2;

    when delay2 =>

       tmp_one <= resize(Y1*a1,tmp_one'high,tmp_one'low);
       tmp_two <= resize(Y2*a2,tmp_two'high,tmp_two'low);
       tmp_three <= resize(Y3*a3,tmp_three'high,tmp_three'low);
       tmp_four <= resize(Y4*a3,tmp_four'high,tmp_four'low);
       tmp_five <= resize(Y5*a5,tmp_five'high,tmp_five'low);

     state <= stop;

    when stop =>

       Y1_delay <= Y1;
       Y2_delay <= Y2;
       Y3_delay <= Y3;
       Y4_delay <= Y4;
       Y5_delay <= Y5;
```

```


        state <= start;
        when others => null;

      end case;
      end if;
  end process delay_process_section;

end rtl;
```

B.6 Pulse Width Modulation source code

Listing B.9: Pulse width modulator

```
-- Design Unit : Pulse width modulation
-- File name   : PWM.vhd
-- Description : Generate pulse width modulation from the output
    of the loop filter. A 7-bit sawtooth wave is generated and
    compared against the output of the loop filter. The PWM output
    is then averaged, ripple compensated and fed back to the
    compensator block.
--
-- Author      : Jason Quibell
--               Centre for Instrumentation Research
--               Cape Peninsula University of Technology
-- Revision    : Version 1.6 13/04/2011

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

Library ieee_proposed;
use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;


--------------------------------------------------------------
entity PWM is
  port (
    clock      : in std_logic;
    filtOut    : in sfixed(1 downto -23);
    upsampled   : in sfixed(0 downto -23);
    filtIn     : out sfixed(1 downto -23);
    pwmOut     : out std_logic;
      );
end PWM;

architecture a of PWM is

  signal sawtooth    : sfixed(1 downto -6):= "11000000"; --  -1

  signal pwmHold     : std_logic;
  signal pwmFix    : sfixed(1 downto -23);
  signal pwmDelay1   : sfixed(1 downto -23) := "
     0000000000000000000000000";
  signal pwmDelay2   : sfixed(1 downto -23) := "
     0000000000000000000000000";
  signal pwmDelay3   : sfixed(1 downto -23) := "
     0000000000000000000000000";

  signal one   : sfixed(1 downto -6);
  signal p63   : sfixed(1 downto -6);
  signal n64   : sfixed(1 downto -6);

  signal PWMpos : sfixed(1 downto -23):= "0100000000000000000000000";
  signal PWMneg : sfixed(1 downto -23):= "1100000000000000000000000";

begin

  process(clock, filtOut, pwmFix)
  begin
  --7bit PWM (8bits are used in calculation for signal compatibility)
  one   <= "00000001";
  p63   <= "00111111";
  n64   <= "11000000";


    if(clock'event and clock = '1') then

      if sawtooth = p63 then
        sawtooth <= n64;
        --Audio is now larger
        pwmOut <= '0';

        --Delays
        pwmDelay1 <= "1100000000000000000000000";
        pwmDelay2 <= pwmDelay1;
        pwmDelay3 <= pwmDelay2;

        --Averaging
        filtIn <= resize(upsampled-((PWMneg+pwmDelay1+pwmDelay2+pwmDelay3)/4)-sawtooth,pwmFix'high,pwmFix'low);

      else
        sawtooth <= resize(sawtooth + one, sawtooth'high,sawtooth'low);
        if filtOut > sawtooth then
          pwmOut <= '1';

          --Delays
          pwmDelay1 <= "0100000000000000000000000"; --    1
          pwmDelay2 <= pwmDelay1;
          pwmDelay3 <= pwmDelay2;

        --Averaging
          filtIn <= resize(upsampled-((PWMpos+pwmDelay1+pwmDelay2+pwmDelay3)/4)-sawtooth,pwmFix'high,pwmFix'low);

        else
          pwmOut <= '0';
          --Delays
          pwmDelay1 <= "1100000000000000000000000";
          pwmDelay2 <= pwmDelay1;
          pwmDelay3 <= pwmDelay2;

          --Averaging

          filtIn <= resize(upsampled-((PWMneg+pwmDelay1+pwmDelay2+
             pwmDelay3)/4)-sawtooth,pwmFix'high,pwmFix'low);
        end if;

      end if;

    end if;
  end process;
end a;
```

Appendix C

Matlab source code

C.1 Interpolation

Listing C.1: Upsample by factor L

```
%Upsample signal by factor L
%Jason Quibell
%June 2008

clear;close all;

%Export fig options
opts = struct('LockAxes',0,'FontMode','fixed','FontSize',8,'height',10,'color','rgb');

fs = 48e3;                          % sampling rate
Ts = 1/fs;
N = 2048;                           % number of samples
n = 0:N-1;
t = n*Ts;
L = 4;                              % up sampling factor
A = 1;                              % amplitude

freq = 1e3;                              % generate a 1 kHz sine wave
signal = A*sin(2*pi*freq*t);


% up sampling by a factor of L with zero-padding
upsampled=zeros(1,L*N);
for n=0:1:N-1
   upsampled(L*n+1)=signal(n+1);
end

%Spectral components
SIGNAL = 2*abs(fft(signal,N))/N;
SIGNAL(1) = SIGNAL(1)/2;

UPSAMPLED = 2*abs(fft(upsampled,N*L))/N*L;
```

```matlab
UPSAMPLED(1) = UPSAMPLED(1)/2;
UPSAMPLED = UPSAMPLED/4;

freqplot = (0:N/2 -1)*fs/N;
freqplot2 = (0:N*L/2 -1)*fs/N;

%Plot waveforms
subplot(2,1,1);
stem(signal);
title('Original sine wave');
xlabel('Samples');
ylabel('Amplitude');
axis([0 length(signal) -1.1 1.1])

subplot(2,1,2);
stem(upsampled);
title('Upsampled sine wave');
xlabel('Samples');
ylabel('Amplitude');
axis([0 length(upsampled) -1.1 1.1])

exportfig(gcf,'upsampleSine.eps',opts);

%plot the freq of sine wave in the top panel
figure
subplot(2,1,1)
plot(freqplot,SIGNAL(1:1:N/2))
title('Original sine wave spectrum')
xlabel('Frequency (Hz)');
ylabel('Amplitude');
%axis([0 length(SIGNAL) -1.1 1.1])

%plot the upsampled sine wave in the bottom panel
subplot(2,1,2)
plot(freqplot2,UPSAMPLED(1:1:length(UPSAMPLED)/2))
title('Upsampled sine wave spectrum')
xlabel('Frequency (Hz)');
ylabel('Amplitude');
%axis([0 length(UPSAMPLED) -1.1 1.1])

exportfig(gcf,'upsampleSpectrum.eps',opts);
```

Listing C.2: Filter upsampled signal

```
%Filter upsampled signal
%Jason Quibell
%June 2008


filtered=filter(Num,1,upsampled);          % apply interpolation filter

filtered = filtered*4;
%Spectral components

UPSAMPLED = 2*abs(fft(upsampled,N*L))/N*L;
UPSAMPLED(1) = UPSAMPLED(1)/2;
UPSAMPLED = UPSAMPLED/4;

FILTERED = 2*abs(fft(filtered,N*L))/N*L;
FILTERED(1) = FILTERED(1)/2;
FILTERED = FILTERED/16;

freqplot2 = (0:N*L/2 -1)*fs/N;

%Plot waveforms
subplot(2,1,1);
stem(upsampled);
title('Upsampled sine wave');
xlabel('Samples');
ylabel('Amplitude');
axis([0 length(upsampled) -1.1 1.1])

subplot(2,1,2);
stem(filtered);
title('Filtered upsampled sine wave');
xlabel('Samples');
ylabel('Amplitude');
axis([0 length(filtered) -1.1 1.1])

exportfig(gcf,'FIRupsAmp.eps',opts);

%plot the freq of sine wave in the top panel
figure;
subplot(2,1,1)
plot(freqplot2,UPSAMPLED(1:1:length(UPSAMPLED)/2))
title('Upsampled sine wave spectrum')
xlabel('Frequency (Hz)');
ylabel('Amplitude');

%plot the upsampled sine wave in the bottom panel
subplot(2,1,2)
plot(freqplot2,FILTERED(1:1:length(FILTERED)/2))
title('Filtered sine wave spectrum')
xlabel('Frequency (Hz)');
```

```
ylabel('Amplitude');
%axis([0 length(UPSAMPLED) -1.1 1.1])

exportfig(gcf,'FIRupsAmpFIlt.eps',opts);
```

Listing C.3: Polyphase filter implementation

```
%Polyphase filter implementation for interpolation
%Jason Quibell
%July 2008


fs = 96e3;                     % sampling rate
Ts = 1/fs;
N = 1024;                      % number of samples
n = 0:N-1;
t = n*Ts;
L = 8;                         % interpolation factor
A = 5;                         % amplitude


freq = 1e3;                     %sine wave frequency
signal = A*sin(2*pi*freq*t);    % generate a sine wave
%noise = 0.1*(randn(1,length(signal))); %Random noise
%signal = signal + noise;

% polyphase filters

for i=1:1:8
    p(i,:)=Num(i:L:length(Num));    %sub filter creation
end

for i=1:1:8
    w(i,:)=filter(p(i,:) ,1,signal); %filtering
end

for i=1:1:8
    y(i,:)=zeros(1,L*length(w(i,:)));
end

for i=1:1:8
    y(i,(i:L:L*(length(w(i,:)))))=w(i,:); %filtering
end

filtered=y(1,:)+y(2,:)+y(3,:)+y(4,:)+y(5,:)+y(6,:)+y(7,:)+y(8,:);
```

Listing C.4: CIC automatic filter design

```
%Design CIC filter and CIC compensator filter
%Jason Quibell
%March 2009

clear; close all;

%Export fig options
opts = struct('LockAxes',0,'FontMode','fixed','FontSize',8,'height',10,'color','rgb');

M = 1; %Differential delay
L = 16; %Oversampling ratio
Fp = 22e3; %Passband frequency
Fs = 3.072e6; %Final sample rate
Ast = 60; %Aliasing attenuation (60 dB)

%CIC filter
f = fdesign.interpolator(L,'CIC',M,Fp,Ast,Fs);
Hm = design(f);
fvtool(Hm);

%CIC compensator
Nsecs = Hm.NumberOfSections;
d = fdesign.ciccomp(M,Nsecs);
Hd = design(d);
fvtool(Hd);

%Cascade filters
Hc = cascade(Hd,Hm);
fvtool(Hc);
```

Listing C.5: CIC filter implementation

```
%2nd order CIC filter implementation
%Jason Quibell
%March 2009

Lcic = 100;
delayBuffer = zeros(1,1);              %Delay of "1"
delayBuffer2 = zeros(1,1);              %Delay of "1"
delayBuffer3 = zeros(1,1);              %Delay of "1"
intOut = 0;
intOut2 = 0;
intOut3 = 0;
filteredx= [];                         %Initialise variable
filtered = reference / (Lcic);         %Compensate gain factor

for ii = 1:length(filtered)



        %Comb
            combOut = filtered(ii) - delayBuffer(end);
            delayBuffer(2:end) = delayBuffer(1:end-1);
            delayBuffer(1) = filtered(ii);

            combOut2 = combOut - delayBuffer2(end);
            delayBuffer2(2:end) = delayBuffer2(1:end-1);
            delayBuffer2(1) = combOut;


        %Upsample
            combOutU = [ combOut2 zeros(1,Lcic-1)];

        %Integrator
            for jj = 0:Lcic-1
                intOut = intOut + combOutU(jj+1);
                intOut2 = intOut2 + intOut;
                filteredx = [filteredx intOut2];
            end
end

upsample = filteredx;
```

C.2 Polynomial interpolation

Listing C.6: Newton's interpolation formula

```
%Newton polynomial interpolation
%Jason Quibell
%May 2009

count   = 0;
xData = [1 2 3 4 5 6];
yData = [0 0 0 0 0 0];
upsample = [];

for z = 1:1:length(reference)

yData(6) = reference(z);

%Newton Coefficients (Divided Differences)
n = length(xData);
coeff = yData;


for k = 2:n
    coeff(k:n) = (coeff(k:n) - coeff(k-1))./(xData(k:n) - xData(k-1));
end

%Interpolation using polynomial coefficients

for x = 4: 0.015625: 4.96875  %64x oversampling
    n = length(xData);
    p = coeff(n);

        for k = 1:n-1;
            p = coeff(n-k) + (x - xData(n-k))*p;
        end

        upsample = [upsample p];
end
yData(1) = yData(2);
yData(2) = yData(3);
yData(3) = yData(4);
yData(4) = yData(5);
yData(5) = yData(6);

end
```

Listing C.7: FPGA ROM file contents generator

```
%Generate MIF ROM file contents for FPGA
%Jason Quibell
%September 2009

clear;
array = [];
x_val1 = [];
x_val2 = [];
x_val3 = [];
x_val4 = [];
x_val5 = [];
xData = [1 2 3 4 5 6];

for x  = 4: 0.015625: 4.984375    % 64X OSR
     x1 = (x-xData(1));
     x2 = (x-xData(1))*(x-xData(2));
     x3 = (x-xData(1))*(x-xData(2))*(x-xData(3));
     x4 = (x-xData(1))*(x-xData(2))*(x-xData(3))*(x-xData(4));
     x5 = (x-xData(1))*(x-xData(2))*(x-xData(3))*(x-xData(4))*(x-
         xData(5));

     array  = [array; x1 x2 x3 x4 x5];
     x_val1 = [x_val1; x1];
     x_val2 = [x_val2; x2];
     x_val3 = [x_val3; x3];
     x_val4 = [x_val4; x4];
     x_val5 = [x_val5; x5];
end

A_sign         = fi(array,1,31,7);    %Fixed point signed
BINARYsign     = bin(A_sign);

A_sign         = fi(x_val1,1,31,7);    %Fixed point signed
BINARYsign1    = bin(A_sign);

A_sign         = fi(x_val2,1,31,7);    %Fixed point signed
BINARYsign2    = bin(A_sign);

A_sign         = fi(x_val3,1,31,7);    %Fixed point signed
BINARYsign3    = bin(A_sign);

A_sign         = fi(x_val4,1,31,7);    %Fixed point signed
BINARYsign4    = bin(A_sign);

A_sign         = fi(x_val5,1,31,7);    %Fixed point signed
BINARYsign5    = bin(A_sign);
```

C.3 Noise shaper filter coefficient calculator

Listing C.8: Coefficient calculation for a 3rd order loop filter

```
%Coefficient Calculation
%3rd Order noise shaper
%Jason Quibell
%December 2010

syms a1 a2 a3 g1 z

%Transfer function of CIFF with resonator feedback

%Y(z) / X(z) =
three = (a1/(z-1)) + ((a2*z)/(z^2+(-2+g1)*z + 1)) ...
       + (1/(z-1))*((a3*z)/(z^2+(-2+g1)*z+1));

[num den] = numden(three);

[b,a] = tfdata(C,'v');

num_z = collect(num);
den_z = collect(den);

a1 = b(4);
a2 = b(2) - a1;
g1 = 3 - a(3);

a3 = b(3) + (b(4)*2) - (a1*g1) + a2;


%Clean up

clear num_z den_z num den  five

%Generate fixed point values of coefficients

a1_sign   = fi(a1,1,35,34);    %35bit Fixed point signed
a1_BIN    = bin(a1_sign);

a2_sign   = fi(a2,1,40,39);    %40bit Fixed point signed
a2_BIN    = bin(a2_sign);

a3_sign   = fi(a3,1,42,41);    %42bit Fixed point signed
a3_BIN    = bin(a3_sign);


g1_sign   = fi(g1,1,40,39);    %40bit Fixed point signed
g1_BIN    = bin(g1_sign);
```

Listing C.9: Coefficient calculation for a 5th order loop filter

```
%Coefficient Calculation
%5th Order noise shaper
%Jason Quibell
%August 2010

syms a1 a2 a3 a4 a5 g1 g2 z x

%Transfer function of CIFF with resonator feedback

%Y(z) / X(z) =
five = (a1/(z-1)) + ((a2*z)/(z^2+(-2+g1)*z + 1)) ...
       + (1/(z-1))*((a3*z)/(z^2+(-2+g1)*z+1)) ...
       + ((a4*z)/(z^2+(-2+g2)*z+1)*(z/(z^2+(-2+g1)*z+1))) ...
       + (1/(z-1)) * (z/(z^2+(-2+g1)*z + 1)) * ((a5*z)/(z^2+(-2+g2)
           *z +1));

[num den] = numden(five);

[b,a] = tfdata(C,'v');

num_z = collect(num);
den_z = collect(den);

%Solve coefficient "a1"
a1 = b(6);

g2_temp = 3*(a(2) +5-g2) + 3*g2 - g2*(a(2) + 5 -g2) - 10;

%INSERT a(4) where -9.999998... is:
%INSERT g2_temp where other stuff is first



%49MHz
%g2_x = solve('g2*(g2 - 2872429995/562949953421312) -
    5629490916923135/562949953421312 = -9.999984692621844');
%g2 = double(g2_x(1));

%24MHz
%g2_x = solve('g2*(g2 - 15914144539/1125899906842624) -
    11258951325992623/1125899906842624 = -9.99995759623830');

%g2 = double(g2_x(1));


%12MHz
g2_x = solve('g2*(g2 - 36060501917/562949953421312) -
    5629391352707369/562949953421312 = -9.99980783163673');

g2 = double(g2_x(1));

```

```
%g1
g1 = double(a(2) + 5 - g2);

%Solve a2:
a2 = b(2) - a1 ;

%Solve a3:
a3 = b(5) + a2 + 4*a1 - a1*g1 - a1*g2;

%Solve a4:
a4 = b(3) - a3 + 3*a2 + 4*a1 - a1*g1 - a1*g2 - a2*g2;

%Solve a5:
a5 = b(4) - 6*a1 - 3*a2 + 2*a3 + a4 + 2*a1*g1 + 2*a1*g2 + a2*g2 - a3*g2 - a1*g1*g2;

%Clean up

%clear x z num_z den_z num den g2_x g2_temp five

%Generate fixed point values of coefficients

a1_sign   = fi(a1,1,35,34);    %35bit Fixed point signed
a1_BIN    = bin(a1_sign);

a2_sign   = fi(a2,1,40,39);    %40bit Fixed point signed
a2_BIN    = bin(a2_sign);

a3_sign   = fi(a3,1,42,41);    %42bit Fixed point signed
a3_BIN    = bin(a3_sign);

a4_sign   = fi(a4,1,45,44);    %45bit Fixed point signed
a4_BIN    = bin(a4_sign);

a5_sign   = fi(a5,1,50,49);    %50bit Fixed point signed
a5_BIN    = bin(a5_sign);

g1_sign   = fi(g1,1,40,39);    %40bit Fixed point signed
g1_BIN    = bin(g1_sign);

g2_sign   = fi(g2,1,40,39);    %40bit Fixed point signed
g2_BIN    = bin(g2_sign);
```

C.4 Modulation

Listing C.10: Pulse width modulation

```
%Uniform Pulse Width Modulation
%Jason Quibell
%June 2009

clear;
carrier=0:1/49152000:1/1000;
reference=0:1/192000:1/1000;
len_c=length(carrier);
len_r=length(reference);


for t=1:1:len_c

    fs=384e3; %Carrier wave frequency: 384 kHz
    f_carrier(t)=(2/pi)*asin(sin(2*pi*fs*carrier(t)))+1;
end

for t = 1:1:len_r

    fr=6.7e3;     %Reference wave frequency
    ref(t)=0.8*sin(2*pi*fr*reference(t))+1; % 0.8 amplitude

    if f_carrier(t)<ref(t)

        switch(t)=1;

    elseif ref(t)<f_carrier(t)
        switch(t)=0;
    end
    end
figure;
plot(carrier,f_c1,'g');
hold on;
plot(reference,f_r,'b');
hold on;
plot(reference,s1,'r');
ylabel('Amplitude');
xlabel('Time [s]');
axis([0 0.0001 -0.1 1.1]);
hold off;
```

Listing C.11: 5th order delta-sigma modulator

```
%Implementation of 5th Order DSM CIFB Structure
%Jason Quibell
%March 2009

fs = 48e3;                       % sampling rate
Ts = 1/fs;
N = 2048;                        % number of samples
m = 1:N;
n = 0:N-1;
t = n*Ts;
L = 128;                           % up sampling factor
A = 0.55;                           % amplitude
k = 1;


 %SDM Coefficients
 a1 = 0.0007;
 a2 = 0.0100;
 a3 = 0.0737;
 a4 = 0.3149;
 a5 = 0.8090;

 b1 = 0.0007;
 b2 = 0.0100;
 b3 = 0.0737;
 b4 = 0.3149;
 b5 = 0.8090;
 b6 = 1;

 y(1) = 0;
 y1(1) = 0;
 y2(1) = 0;
 y3(1) = 0;
 y4(1) = 0;
 y5(1) = 0;

 v(1) = 0;

freq = 10e3;                           % generate a sine wave
u = A*sin(2*pi*freq*t);

 % up sampling by a factor of L
upsampled=zeros(1,L*(length(u)));

for n=0:1:(length(u)-1)
   upsampled(L*n+1)=u(n+1);
end

B=fir1(256,1/(L*2));
u=filter(B,1,v);              % apply interpolation filter

```

```


 for n = 2:length(u)-1

 % ------ One-bit quantiser

     if (y(n-1) >= 0)   v(n) = +k;
     else               v(n) = -k;
     end

     e1(n) = (b1*u(n)) - (a1*v(n));
     y1(n) = y1(n-1) + e1(n);

     e2(n) = b2*u(n) - a2*v(n);
     y2(n) = y1(n) + y2(n-1) + e2(n);

     e3(n) = b3*u(n) - a3*v(n);
     y3(n) = y2(n) + y3(n-1) + e3(n);

     e4(n) = b4*u(n) - a4*v(n);
     y4(n) = y3(n) + y4(n-1) + e4(n);

     e5(n) = b5*u(n) - a5*v(n);
     y5(n) = y4(n) + y5(n-1) + e5(n);

     y(n) = y5(n) + b6*u(n);

 end

figure;
pwelch(v,[],[],[],fs*L);

figure;
B=fir1(256,1/(L*2));
filtered=filter(B,1,v);            % apply filter
plot(filtered)
figure;
pwelch(filtered,[],[],[],fs*L);
```

C.5 Spectral content

Listing C.12: Spectral content of PWM waveform

```
%FFT of PWM waveform
%Jason Quibell
%November 2009


figure();

fs = 98.304e6/2;
Y = fft(PWM.*blackman(length(PWM)))/(length(PWM)*0.2);
hz500k = 500000*length(Y)/fs; %Limit viewing to 500kHz
f=(0:hz500k)*fs/length(Y);
plot(f,20*log10(abs(Y(1:length(f)))));
ylabel('Magnitude (dB)');
xlabel('Frequency (Hz)');
```

Appendix D

Data sheets

D.1 SRC4392

Burr-Brown Products
from Texas Instruments

SRC4392

SBFS029C–DECEMBER 2005–REVISED SEPTEMBER 2007

Two-Channel, Asynchronous Sample Rate Converter with Integrated Digital Audio Interface Receiver and Transmitter

FEATURES

- **Two-Channel Asynchronous Sample Rate Converter (SRC)**
  - **Dynamic Range with –60dB Input (A-Weighted): 144dB typical**
  - **Total Harmonic Distortion and Noise (THD+N) with Full-Scale Input: –140dB typical**
  - **Supports Audio Input and Output Data Word Lengths Up to 24 Bits**
  - **Supports Input and Output Sampling Frequencies Up to 216kHz**
  - **Automatic Detection of the Input-to-Output Sampling Ratio**
  - **Wide Input-to-Output Conversion Range: 16:1 to 1:16 Continuous**
  - **Excellent Jitter Attenuation Characteristics**
  - **Digital De-Emphasis Filtering for 32kHz, 44.1kHz, and 48kHz Input Sampling Rates**
  - **Digital Output Attenuation and Mute Functions**
  - **Output Word Length Reduction**
  - **Status Registers and Interrupt Generation for Sampling Ratio and Ready Flags**
- **Digital Audio Interface Transmitter (DIT)**
  - **Supports Sampling Rates Up to 216kHz**
  - **Includes Differential Line Driver and CMOS Buffered Outputs**
  - **Block-Sized Data Buffers for Both Channel Status and User Data**
  - **Status Registers and Interrupt Generation for Flag and Error Conditions**
- **User-Selectable Serial Host Interface: SPI or Philips I^{2}C™**
  - **Provides Access to On-Chip Registers and Data Buffers**

U.S. Patent No. 7,262,716

- **Digital Audio Interface Receiver (DIR)**
  - **PLL Lock Range Includes Sampling Rates from 20kHz to 216kHz**
  - **Includes Four Differential Input Line Receivers and an Input Multiplexer**
  - **Bypass Multiplexer Routes Line Receiver Outputs to Line Driver and Buffer Outputs**
  - **Block-Sized Data Buffers for Both Channel Status and User Data**
  - **Automatic Detection of Non-PCM Audio Streams (DTS CD/LD and IEC 61937 formats)**
  - **Audio CD Q-Channel Sub-Code Decoding and Data Buffer**
  - **Status Registers and Interrupt Generation for Flag and Error Conditions**
  - **Low Jitter Recovered Clock Output**
- **Two Audio Serial Ports (Ports A and B)**
  - **Synchronous Serial Interface to External Signal Processors, Data Converters, and Logic**
  - **Slave or Master Mode Operation with Sampling Rates up to 216kHz**
  - **Supports Left-Justified, Right-Justified, and Philips I^{2}S™ Data Formats**
  - **Supports Audio Data Word Lengths Up to 24 Bits**
- **Four General-Purpose Digital Outputs**
  - **Multifunction Programmable Via Control Registers**
- **Extensive Power-Down Support**
  - **Functional Blocks May Be Disabled Individually When Not In Use**
- **Operates From +1.8V Core and +3.3V I/O Power Supplies**
- **Small TQFP-48 Package, Compatible with the SRC4382 and DIX4192**

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this data sheet.

APPLICATIONS

- DIGITAL AUDIO RECORDERS AND MIXING DESKS
- DIGITAL AUDIO INTERFACES FOR COMPUTERS
- DIGITAL AUDIO ROUTERS AND DISTRIBUTION SYSTEMS
- BROADCAST STUDIO EQUIPMENT
- DVD/CD RECORDERS
- SURROUND SOUND DECODERS AND A/V RECEIVERS
- CAR AUDIO SYSTEMS

DESCRIPTION

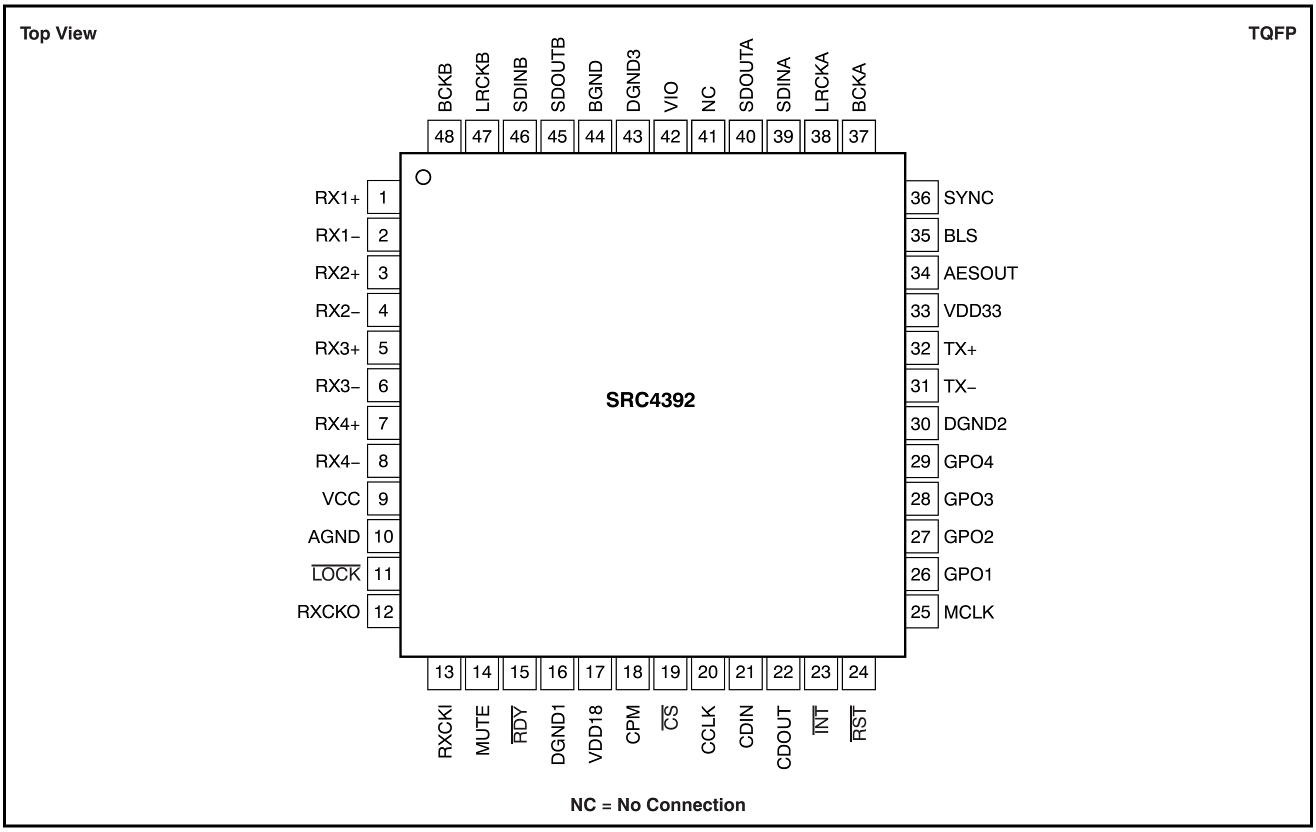
The SRC4392 is a highly-integrated CMOS device designed for use in professional and broadcast digital audio systems. The SRC4392 combines a high-performance, two-channel, asynchronous sample rate converter (SRC) with a digital audio interface receiver (DIR) and transmitter (DIT), two audio serial ports, and flexible distribution logic for interconnection of the function block data and clocks.

The DIR and DIT are compatible with the AES3, S/PDIF, IEC 60958, and EIAJ CP-1201 interface standards. The audio serial ports, DIT, and SRC may be operated at sampling rates up to 216kHz. The DIR lock range includes sampling rates from 20kHz to 216kHz.

The SRC4392 is configured using on-chip control registers and data buffers, which are accessed through either a 4-wire serial peripheral interface (SPI) port, or a 2-wire Philips I^{2}C bus interface. Status registers provide access to a variety of flag and error bits, which are derived from the various function blocks. An open drain interrupt output pin is provided, and is supported by flexible interrupt reporting and mask options via control register settings. A master reset input pin is provided for initialization by a host processor or supervisory functions.

The SRC4392 requires a +1.8V core logic supply, in addition to a +3.3V supply for powering portions of the DIR, DIT, and line driver and receiver functions. A separate logic I/O supply supports operation from +1.65V to +3.6V, providing compatibility with low voltage logic interfaces typically found on digital signal processors and programmable logic devices. The SRC4392 is available in a lead-free, TQFP-48 package, and is pin- and register-compatible with the Texas Instruments SRC4382 and DIX4192 products.

PIN CONFIGURATION



PIN DESCRIPTIONS

NAME	PIN NUMBER	I/O	DESCRIPTION
RX1+	1	Input	Line Receiver 1, Noninverting Input
RX1–	2	Input	Line Receiver 1, Inverting Input
RX2+	3	Input	Line Receiver 2, Noninverting Input
RX2–	4	Input	Line Receiver 2, Inverting Input
RX3+	5	Input	Line Receiver 3, Noninverting Input
RX3–	6	Input	Line Receiver 3, Inverting Input
RX4+	7	Input	Line Receiver 4, Noninverting Input
RX4–	8	Input	Line Receiver 4, Inverting Input
VCC	9	Power	DIR Comparator and PLL Power Supply, +3.3V Nominal
AGND	10	Ground	DIR Comparator and PLL Power-Supply Ground
$\overline{\text{LOCK}}$	11	Output	DIR PLL Lock Flag (active Low)
RXCKO	12	Output	DIR Recovered Master Clock (tri-state output)
RXCKI	13	Input	DIR Reference Clock
MUTE	14	Input	SRC Output Mute (active High)
$\overline{\text{RDY}}$	15	Output	SRC Ready Flag (active Low)
DGND1	16	Ground	Digital Core Ground
VDD18	17	Power	Digital Core Supply, +1.8V Nominal
CPM	18	Input	Control Port Mode, 0 = SPI Mode, 1 = I^{2}C Mode
$\overline{\text{CS}}$ or A0	19	Input	Chip Select (active Low) for SPI Mode or Programmable Slave Address for I^{2}C Mode
CCLK or SCL	20	Input	Serial Data Clock for SPI Mode or I^{2}C Mode
CDIN orA1	21	Input	SPI Port Serial Data input or Programmable Slave Address for I^{2}C Mode
CDOUT or SDA	22	I/O	SPI Port Serial Data Output (tri-state output) or Serial Data I/O for I^{2}C Mode

PIN DESCRIPTIONS (continued)

NAME	PIN NUMBER	I/O	DESCRIPTION
$\overline{\text{INT}}$	23	Output	Interrupt Flag (open-drain, active Low)
$\overline{\text{RST}}$	24	Input	Reset (active Low)
MCLK	25	Input	Master Clock
GPO1	26	Output	General-Purpose Output 1
GPO2	27	Output	General-Purpose Output 2
GPO3	28	Output	General-Purpose Output 3
GPO4	29	Output	General-Purpose Output 4
DGND2	30	Ground	DIR Line Receiver Bias and DIT Line Driver Digital Ground
TX–	31	Output	DIT Line Driver Inverting Output
TX+	32	Output	DIT Line Driver Noninverting Output
VDD33	33	Power	DIR Line Receiver Bias and DIT Line Driver Supply, +3.3V Nominal
AESOUT	34	Output	DIT Buffered AES3-Encoded Data
BLS	35	I/O	DIT Block Start Clock
SYNC	36	Output	DIT internal Sync Clock
BCKA	37	I/O	Audio Serial Port A Bit Clock
LRCKA	38	I/O	Audio Serial Port A Left/Right Clock
SDINA	39	Input	Audio Serial Port A Data Input
SDOUTA	40	Output	Audio Serial Port A Data Output
NC	41	—	No Internal Signal Connection, Internally Bonded to ESD Pad
VIO	42	Power	Logic I/O Supply, +1.65V to +3.6V
DGND3	43	Ground	Logic I/O Ground
BGND	44	Ground	Substrate Ground, Connect to AGND (pin 10)
SDOUTB	45	Output	Audio Serial Port B Data Output
SDINB	46	Input	Audio Serial Port B Data Input
LRCKB	47	I/O	Audio Serial Port B Left/Right Clock
BCKB	48	I/O	Audio Serial Port B Bit Clock

PRODUCT OVERVIEW

The SRC4392 is a two-channel asynchronous sample rate converter (SRC) with an integrated digital audio interface receiver and transmitter (DIR and DIT). Two audio serial ports, Port A and Port B, support input and output interfacing to external data converters, signal processors, and logic devices. On-chip routing logic provides for flexible interconnection between the five functional blocks. The audio serial ports, DIT, and SRC may be operated at sampling rates up to 216kHz. The DIR is specified for a PLL lock range that includes sampling rates from 20kHz to 216kHz. All function blocks support audio data word lengths up to 24 bits.

The SRC4392 requires an external host processor or logic for configuration control. The SRC4392 includes a user-selectable serial host interface, which operates as either a 4-wire serial peripheral interface (SPI) port or a 2-wire Philips I^{2}C bus interface. The SPI port operates at bit rates up to 40MHz. The I^{2}C bus interface may be operated in standard or fast modes, supporting operation at 100kbps and 400kbps, respectively. The SPI and I^{2}C interfaces provide access to internal control and status registers, as well as the buffers utilized for the DIR and DIT channel status and user data.

The asynchronous SRC is based upon the successful SRC4192 core from Texas Instruments. The SRC in the SRC4392 has been further enhanced to provide exceptional jitter attenuation characteristics, helping to improve overall application performance. The SRC operates over a wide input-to-output sampling ratio range, from 1:16 to 16:1 continuous. The input-to-output sampling ratio is determined automatically by the SRC rate estimation circuitry, with the digital re-sampler parameters being updated in real-time without the need for programming. Interpolation and decimation filter delay are user-selectable. Additional SRC features include de-emphasis filtering, output word length reduction, output attenuation and muting, and input-to-output sampling ratio readback via status registers.

The digital interface receiver (DIR) includes four differential input line receiver circuits, suitable for balanced or unbalanced cable interfaces. Interfacing to optical receiver modules and CMOS logic devices is also supported. The outputs of the line receivers are connected to a 1-of-4 data selector, referred to as the receiver input multiplexer, which is utilized to select one of the four line receiver outputs for processing by the DIR core. The outputs of the line receivers are also connected to a second data selector, the bypass multiplexer, which may be used to route input data streams to the DIT CMOS output buffer and differential line driver functions. This configuration provides a bypass signal path for AES3-encoded input data streams.

The DIR core decodes the selected input stream data and separates the audio, channel status, user, validity, and parity data. Channel status and user data is stored in block-sized buffers, which may be accessed via the SPI or I^{2}C serial host interface, or routed directly to the general-purpose output pins (GPO1 through GPO4). The validity and parity bits are processed to determine error status. The DIR core recovers a low jitter master clock, which may be utilized to generate word and bit clocks using on-chip or external logic circuitry.

The digital interface transmitter (DIT) encodes digital audio input data into an AES3-formatted output data stream. Two DIT outputs are provided, including a differential line driver and a CMOS output buffer. Both the line driver and buffer include 1-of-2 input data selectors, which are utilized to choose either the output of the DIT AES3 encoder, or the output of the bypass multiplexer. The line driver output is suitable for balanced or unbalanced cable interfaces, while the CMOS output buffer supports interfacing to optical transmitter modules and external logic or line drivers. The DIT includes block-sized data buffers for both channel status and user data. These buffers are accessed via either the SPI or I^{2}C host interface, or may be loaded directly from the DIR channel status and user data buffers.

The SRC4392 includes four general-purpose digital outputs, or GPO pins. The GPO pins may be configured as simple logic outputs, which may be programmed to either a low or high state. Alternatively, the GPO pins may be connected to one of 14 internal logic nodes, allowing them to serve as functional, status, or interrupt outputs. The GPO pins provide added utility in applications where hardware access to selected internal logic signals may be necessary.

D.2 Altera Cyclone III (EP3C25)

1. Cyclone III Device Family Overview

CIII51001-2.2

Cyclone® III device family offers a unique combination of high functionality, low power and low cost. Based on Taiwan Semiconductor Manufacturing Company (TSMC) low-power (LP) process technology, silicon optimizations and software features to minimize power consumption, Cyclone III device family provides the ideal solution for your high-volume, low-power, and cost-sensitive applications. To address the unique design needs, Cyclone III device family offers the following two variants:

- Cyclone III: lowest power, high functionality with the lowest cost
- Cyclone III LS: lowest power FPGAs with security

With densities ranging from 5K to 200K logic elements (LEs) and 0.5 Mbits to 8 Mbits of memory for less than ¼ watt of static power consumption, Cyclone III device family makes it easier for you to meet your power budget. Cyclone III LS devices are the first to implement a suite of security features at the silicon, software, and intellectual property (IP) level on a low-power and high-functionality FPGA platform. This suite of security features protects the IP from tampering, reverse engineering and cloning. In addition, Cyclone III LS devices support design separation which enables you to introduce redundancy in a single chip to reduce size, weight, and power of your application.

This chapter contains the following sections:

Cyclone III Device Family Features

Cyclone III device family offers the following features:

Lowest Power FPGAs

- Lowest power consumption due to:
  - TSMC low-power process technology
  - Altera® power-aware design flow
- Low-power operation offers the following benefits:
  - Extended battery life for portable and handheld applications
  - Reduced or eliminated cooling system costs
  - Operation in thermally-challenged environments
- Hot-socketing operation support

Design Security Feature

Cyclone III LS devices offer the following design security features:

- Configuration security using advanced encryption standard (AES) with 256-bit volatile key
- Routing architecture optimized for design separation flow with the Quartus® II software
  - Design separation flow achieves both physical and functional isolation between design partitions
- Ability to disable external JTAG port
- Error Detection (ED) Cycle Indicator to core
  - Provides a pass or fail indicator at every ED cycle
  - Provides visibility over intentional or unintentional change of configuration random access memory (CRAM) bits
- Ability to clear contents of the FPGA logic, CRAM, embedded memory, and AES key
- Internal oscillator enables system monitor and health check capabilities

Increased System Integration

- High memory-to-logic and multiplier-to-logic ratio
- High I/O count, low-and mid-range density devices for user I/O constrained applications
  - Adjustable I/O slew rates to improve signal integrity
  - Supports I/O standards such as LVTTL, LVCMOS, SSTL, HSTL, PCI, PCI-X, LVPECL, bus LVDS (BLVDS), LVDS, mini-LVDS, RSDS, and PPDS
  - Supports the multi-value on-chip termination (OCT) calibration feature to eliminate variations over process, voltage, and temperature (PVT)
- Four phase-locked loops (PLLs) per device provide robust clock management and synthesis for device clock management, external system clock management, and I/O interfaces
  - Five outputs per PLL
  - Cascadable to save I/Os, ease PCB routing, and reduce jitter
  - Dynamically reconfigurable to change phase shift, frequency multiplication or division, or both, and input frequency in the system without reconfiguring the device
- Remote system upgrade without the aid of an external controller
- Dedicated cyclical redundancy code checker circuitry to detect single-event upset (SEU) issues
- Nios® II embedded processor for Cyclone III device family, offering low cost and custom-fit embedded processing solutions

- Wide collection of pre-built and verified IP cores from Altera and Altera Megafunction Partners Program (AMPP) partners
- Supports high-speed external memory interfaces such as DDR, DDR2, SDR SDRAM, and QDRII SRAM
  - Auto-calibrating PHY feature eases the timing closure process and eliminates variations with PVT for DDR, DDR2, and QDRII SRAM interfaces

Cyclone III device family supports vertical migration that allows you to migrate your device to other devices with the same dedicated pins, configuration pins, and power pins for a given package-across device densities. This allows you to optimize device density and cost as your design evolves.

Table 1–1 lists Cyclone III device family features.

Table 1–1. Cyclone III Device Family Features

Family	Device	Logic Elements	Number of M9K Blocks	Total RAM Bits	18 x 18 Multipliers	PLLs	Global Clock Networks	Maximum User I/Os
Cyclone III	EP3C5	5,136	46	423,936	23	2	10	182
	EP3C10	10,320	46	423,936	23	2	10	182
	EP3C16	15,408	56	516,096	56	4	20	346
	EP3C25	24,624	66	608,256	66	4	20	215
	EP3C40	39,600	126	1,161,216	126	4	20	535
	EP3C55	55,856	260	2,396,160	156	4	20	377
	EP3C80	81,264	305	2,810,880	244	4	20	429
	EP3C120	119,088	432	3,981,312	288	4	20	531
Cyclone III LS	EP3CLS70	70,208	333	3,068,928	200	4	20	413
	EP3CLS100	100,448	483	4,451,328	276	4	20	413
	EP3CLS150	150,848	666	6,137,856	320	4	20	413
	EP3CLS200	198,464	891	8,211,456	396	4	20	413