Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) Major Qualifying Projects

April 2009

DSP Modulated Class D Audio Amplifier

Craig Ropi
Worcester Polytechnic Institute

Jameson John Collins
Worcester Polytechnic Institute

Wadii Bellamine
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation
Ropi, C., Collins, J. J., & Bellamine, W. (2009). *DSP Modulated Class D Audio Amplifier*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/2250

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

DSP Modulated Class D Audio Amplifier

A Major Qualifying Project

Submitted to the Faculty
of
WORCESTER POLYTECHNIC INSTITUTE
in Partial Fulfillment of the requirements for the
Degree of Bachelor of Science
in
Electrical and Computer Engineering

Submitted on April 29, 2009

Sponsoring Agency:	New England Center for Analog & Mixed Signal Design
Submitted by:	Wadii Bellamine Jameson Collins Craig Ropi
Advised by:	Professor Andrew Klein

Abstract

The goal of this project was to create an 80W, 95% efficient Class D audio amplifier with less than 0.5% harmonic distortion and greater than 100 dB zero-input signal to noise ratio that accepts digital inputs. The amplifier that was built comprised of a three-state digital modulator, an H-bridge amplifier, and a passive filter and was capable of accepting both digital and analog audio inputs by means of the SPDIF protocol and an ADC. To allow the modulator design to be quickly altered, it was implemented on a DSP. Because the modulator could be easily changed, several different modulation schemes were simulated, designed and tested in order to achieve optimal audio quality and efficiency results.

Table of Contents

List of Figures

List of Tables

Executive Summary

Class A, B, and AB amplifiers operate in slightly different ways, but none of them can achieve power efficiencies above that of the Class D amplifier. The reason for this is that Class D amplifiers modulate their input in order to make it incredible efficient to amplify the signal. Having efficiencies this high make Class D amplifiers a perfect fit for any application that demands a compact size and low power consumption. However, with Class D amplifiers, increasing sound quality means increasing the complexity of the modulation system. In order to combat this need of increased complexity, the modulator block for this project was built on a digital signals processor (DSP). Using a digital signals processor instead of analog components reduced the increased complexity problem from adding op-amps and flip-flops to simply adding coefficients to an already existing software based IIR filter. Designing the modulator on a digital signals processor also allowed for the amplifier to have digital and analog inputs with almost no additional hardware. Being able to accept a digital input was one of the goals of this project along with achieving 80 watts of output power, 95% power efficiency, over 100 dB of zero-input SNR and less than 0.5% total harmonic distortion. This report discusses the design of a Class D amplifier with that means the previously mentioned goals and all research that was done to build the final working prototype.

The modulation technique chosen for the modulation stage was delta sigma modulation. This method was chosen over pulse width modulation because of delta sigma's ability to shape the modulator's noise transfer function in such a way that would be advantageous for an audio application. Instead of choosing a specific modulator order and modulation frequency, we built the modulator in the DSP in such a way that both could be changed at any given time, until the processing limits of the DSP were reached. Implementing the modulator with software allowed us to first build several simulations of the modulator in MATLAB. Eventually a replica of the DSP's

code was built in MATLAB to allow our team to troubleshoot and change the code with ease. These simulations were essential to our project's success, because they gave our team confidence that our system would meet our signal quality goals. The modulator was chosen to have a three state output so that it could correctly interface with the already designed three state power stage.

The power and filter stages chosen for our project were designed and built by the 2008 NECAMSID Class D audio amplifier MQP group. The power stage consisted of a three state H-bridge amplifier. The filter used was a two pole Butterworth filter. Both the power stage and the filter were built on one printed circuit board and generously leant to us for the duration of our project. The reason for the reuse of this portion of the project was the impressive efficiency that was seen by last year's design.

One of the major advantages of building the modulator of this project on a DSP was the ability to switch the order of the modulator as well as its modulation frequency in order to compare the effectiveness of various combinations. We chose to test modulator orders one through five, with modulation frequencies of 1 MHz, 1.5 MHz, and 2 MHz for a range of input frequencies in the audio band. The large amount of testing that was needed forced our team to automate the sound quality testing process. We did just that by having the DSP internally and automatically create all test signals and to have MATLAB do all the necessary calculations on the recorded outputs.

The results of the audio quality tests met all of projects goals. The project achieved over 105 dB of zero input SNR and less than 0.5% total harmonic distortion. The project's power and efficiency goals were met as well by producing over 90 Watts of output power and consistently achieving over 95% power efficiency.

1. Introduction

Audio amplifiers have long been plagued by trade-offs between size, efficiency, and performance. Traditionally, high performance audio amplifiers have come with large footprints to make room for their heavy heat sinks. While efficient, low heat amplifiers have been relegated to portable devices. The reason for this is that in portable devices, the desire for clarity is pushed aside by the need to retain a small package.

There are several reasons why higher efficiency amplifiers are, and will continue to be in demand. One reason is the increasing power consumption of home entertainment systems. This is due to increasing screen size, the increasing number of multimedia accessories and the increasing cost of energy. However, for many applications the efficiency of an amplifier alone is not enough to make it competitive. For an audio application, the amplifier must also be able to deliver audio clarity comparable to less efficient competitors. As more multimedia accessories are added, power usage can become a concern, second to the concerns of audio quality and a large number of output channels. At one point stereo outputs were considered luxury, but now it is common to see audio sources like DVD players and game consoles outputting 6 or 8 channels of audio. These devices rely primarily on digital audio standards, where several channels can be transmitted over a single cable.

Class D amplifiers have higher efficiencies when compared to traditional linear amplifiers because of their switching nature. In Class D operation, an incoming signal is converted to a digital signal, amplified, and then filtered. Amplification of a digital signal is an efficient process because the transistors used are not operated in their wasteful triode region. The efficiency of these amplifiers implies that their heat dissipation is low; heat sinks can be eliminated from the power devices, reducing overall system size and cost. Most Class D amplifiers are designed with space, heat, and power consumption in mind at the expense of audio quality.

Audio quality does not have to suffer if proper care is given in the creation of the amplifier. High audio quality can be achieved by creating an accurate digital representation of the input signal, and also by combating the major sources of audio distortion through the use of closed loop full system feedback.

1.1 Project Goals

This main goal of this project was to create a prototype of a Class D amplifier and present its function to the NECAMSID lab. The target specifications of the amplifier are shown below in table 1.

Specification	Goal
Input	Digital and Analog
Output Power	80 Watts
Efficiency	> 95%
Zero Input Signal to Noise Ratio	>100dB
Total Harmonic Distortion	<0.5%

Table 1 - Project Specifications

1.2 Report Organization

This MQP progressed in a different fashion than many other EE projects. Because the project was based on signals, and digital processing, much of the work was performed in simulation. The transition from simulation to code is relatively short when compared to the length of the project, so revisions were created daily. This heavily influenced the organization of the report, which does not track the projects progress through revisions, but instead details the process by which simulation had a daily impact on the design.

This report is organized into several large sections. The first section is background information. This section discusses the concepts behind Class D amplifiers and modulators, basics of DSP criteria, and an overview of audio quality standards. The second section discusses the process by which simulations were created and used to synthesize parameters and fine tune modulator code. The final two sections discuss the methods by which test data was accumulated and the results of the testing process.

2. Background

This chapter supplies an explanation of topics that are necessary in order to understand the amplifier's design. The topics discussed are types of amplifiers, digital audio inputs, modulation, and a number of topics from last year's project and report.

2.1 Types of Amplifiers

Before we delve into the intricacies of Class D amplifiers, it is important to explore other types of amplifiers in order to gain a general understanding of what makes an amplifier more or less efficient. A comparison between these amplifiers and Class D amplifiers will then allow us to see why the latter is the best choice for maximizing efficiency. The amplifiers we will explore are the most commonly found in industry: Class A, B, and AB, although other less common types exist such as C, E, F, and GH.

2.1.1 Type A, B, and AB Amplifiers

Class A amplifiers are the simplest, but least efficient type. They consist of an output transistor to amplify the input signal and are therefore conducting 100% of the time, even when there is no input signal. A representation of a Class A amplifier is given in Figure 1.

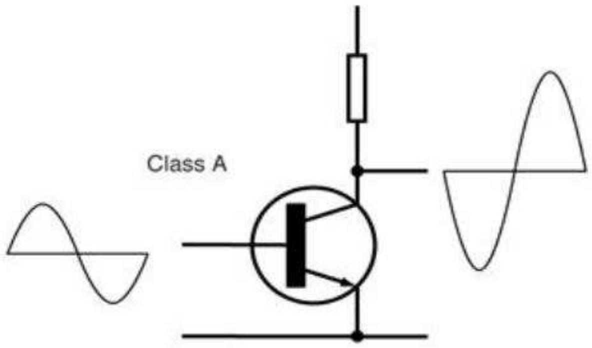


Figure 1 - Class A Amplifier
Figure from Wikipedia[7]

This setup is inherently as linear as possible, allowing for minimal distortion, and therefore the best audio quality. However, because the transistor is conducting all the time, power is drawn all the time, and a great amount of heat is dissipated, requiring large and expensive heat sinks for high power applications. The maximum theoretical efficiency for this type of amplifier is 50% for inductive output coupling, and 25% for capacitive coupling.

Class B amplifiers are similar to Class A, but each half of the Class B device conducts for only half the sinusoidal cycle and is off during the second half. Class B devices are usually setup in a push-pull configuration such that one device conducts during the positive half cycle and the other for the negative half cycle. The output of both devices is then combined to form a single output. This behavior is shown in Figure 2 below.

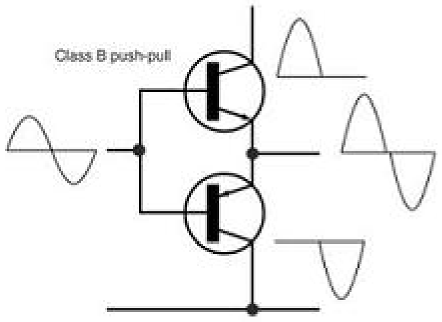


Figure 2 - Class B Push-Pull Configuration
Figure from Wikipedia[7]

Because of this setup, Class B amplifiers are more efficient than Class A because if there is no input signal, there is no current flow at the output. However, the transistors are still dissipating a significant amount heat when they are conducting, and some linearity is compromised due to the distortion at the crossover point while switching from one device to the other. The maximum theoretical efficiency for Class B is 78.5%.

Class AB amplifiers are the same as Class B, except for the fact that each transistor is biased to allow a portion of the signal to pass after its half cycle is over. This solves the issue of crossover distortion, but compromises some efficiency by doing so. Class AB amplifiers are therefore more efficient than Class A, but less efficient that Class B, with a maximum theoretical efficiency of less than 75.8% (typically 50%).

The amplifiers mentioned thus far are all linear amplifiers. Class D amplifiers are of a different nature: they take advantage of switching to maximize efficiency. Class D amplifiers modulate the input signal to convert it into a pulse code format which consists of two values: high and low. This allows Class D amplifiers to take advantage of MOSFETS which act as switches, allowing or disallowing the passage of power to the output stage. Because MOSFETS consume very little power when operating in their triode regions (which are the only regions they need to operate at due to the pulse code signal at their gate), very high efficiencies are possible. Class D amplifiers are able to attain over 95% efficiency. This, in turn, means that virtually no heat sinks are required, reducing the size and cost of Class D amplifiers. Nevertheless, Class D amplifiers suffer from higher distortion than the linear amplifiers mentioned before, because of the noise introduced by the switching, the quantization of the linear input signal, and possible phase delays introduced at every stage. For this reason, Class D amplifiers tend to be more complicated because additional circuitry is required to compensate for distortion.

The following table summarizes and compares the power amplifier types mentioned in this section:

	Class A	Class B	Class AB	Class D
Type	Linear	Linear	Linear	Switching
Advantages	-Least distortion	-More efficient than class A	-Non-linearity of class B solved -Inefficiency of class A overcome	-Highest efficiency -Smallest size -Lowest cost
Disadvantages	-Least efficiency -Large heat sinks	-Non-linearity at crossover	-Less efficient than class B	- Most distortion **or** -complicated design to overcome distortion.
Efficiency (η)	η=28% or 50%	η = 78.5%	50%<η<78.5%	η>=95%

Table 2 - Comparison of Amplifiers

2.1.2 Class D amplifiers: a conceptual description

Class D amplifiers consist of three stages: a modulation stage, a power stage, and a demodulation stage. With this in mind, consider the following figure:

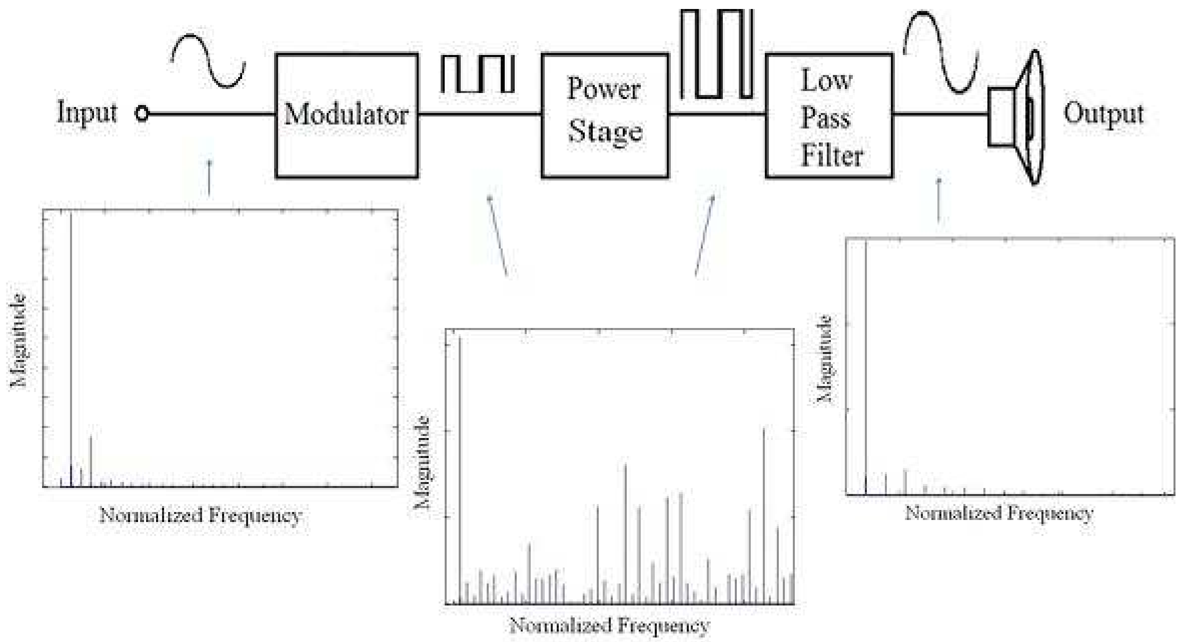


Figure 3 - Class D Stages and Their Frequency Domain Representation

Let us assume for now that the amplifier takes a continuous-time analog input signal. Let us then suppose that this input is a simple sine wave, which will have a single frequency, as shown in the leftmost frequency plot. The modulation stage's role is to convert this signal into a digital pulse code, which in the model above is a two-state pulse code. There are several modulation techniques that can be used to accomplish this, but we only need to quantize the signal in order to represent it as a square wave with pulses of varying densities. The modulation stage therefore adds two types of high frequency noise: one is the noise produced by the quantization of the input signal, and the other is inherent from the frequency spectrum of any square wave. A square wave is made up of a fundamental frequency and all odd harmonics, theoretically to infinity. This explains why the frequency spectrum shown in the middle of Figure 3 consists of the fundamental frequency of the input signal, along with a range of additional high frequencies.

The power stage then takes this pulse code output from the modulator and amplifies it to the desired power. Ideally, this will only increase the magnitudes of all frequencies if we look at the frequency representation of the output of the power stage. However, the quality and switching speed of the MOSFETS used in the power stage is not perfect, and may add additional high frequency noise.

The final step is to remove all the added high frequency noise, which is accomplished by a low-pass filter. The low-pass filter attenuates all undesired frequencies, but leaves all the frequencies in the pass-band untouched. Ideally, if all added frequencies were outside the audible range, only the input's frequencies would remain. In the example above, the single frequency of the original signal remains, so an amplified version of the input is received at the output.

2.2 Modulation

The modulation stage of a Class D amplifier is a critical block with respect to audio quality, because the modulation stage can induce large amounts of noise and harmonic distortion if not designed carefully. At this stage, the choices regarding the over sampling ratio and the order of the modulator determine the effective number of bits created by the modulator, as well as the expected signal to noise ratio. Care must be taken during the design of the modulator to ensure that the digital signal that is created contains the correct frequency information from the input, and that there is minimal harmonic distortion and attenuation. The modulation stage must also be designed with the proposed power stage in mind. Sampling frequencies of the modulator need to be high enough to shape noise out of the signal band but low enough to remain in the efficient regions of the power components.

2.2.1 Pulse Width Modulation

In pulse width modulation (PWM), the input signal is represented by a digital data stream where the duty cycle of the output signal is proportional to the input signal amplitude. There are various ways to implement this modulation technique but the most common method uses a high speed comparator and a ramp signal that acts as a carrier for the output. The input signal is compared to the ramp; the output of the comparator is the digital signal. The frequency of the carrier signal needs to be at least twice that of the input signal, but a typical carrier frequency is around 500 kHz.

While PWM is easy to implement, is used in a variety of systems, highly commercially available, and offers excellent SNR, it has serious draw backs that limit its use in a high fidelity audio amplifier. The noise in the modulated signal has high power in narrow frequency ranges, such as the carrier frequency. Removing these bands of high power noise requires a complex filtering stage.

2.2.2 Delta Sigma Modulation

Delta Sigma modulation uses a technique known as pulse density modulation to encode high resolution data into a low resolution bit stream (the modulated signal). Pulse density modulation represents the amplitude of an analog signal with the relative density of the output pulses. In a continuous time representation of a delta sigma modulator, an input signal is first integrated. At the sampling frequency, Fs, the integration is compared to a predetermined threshold, which either flips or resets the output. Because the output can only be changed at interval 1/Fs the width of the digital pulses are always the same. Even in the most basic Delta Sigma modulators, closed loop feedback is used to monitor the accuracy of the output. The continuous and discrete time models of a first order Delta Sigma modulator are depicted below in Figure 4.

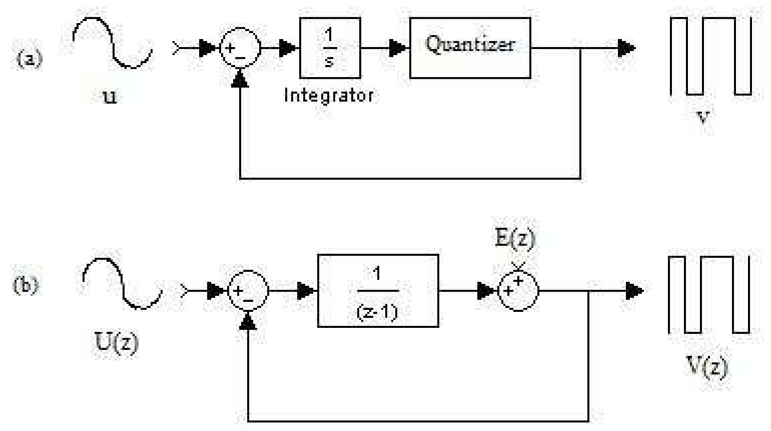


Figure 4 - (a) Continuous Time ΔΣ and (b) Linear Z-Domain Model.

When higher order delta sigma modulators are built, many scale factors can be added in order to manipulate the modulator's performance. A Sample modulator structure, called Cascade-of-integrators, feedback form (CIFB), is shown in Figure 5. In this diagram, z is the discrete time variable z, n is the order of the modulator, and all a's, b's, c's, and g's are constant gains. The "integrators" seen in this model are actually integrators with inherent delays. Basically, each 1/(z-1) block contains a discrete time integrator preceded by a one unit delay. This form was one of the ones used in the process of designing our final modulator.

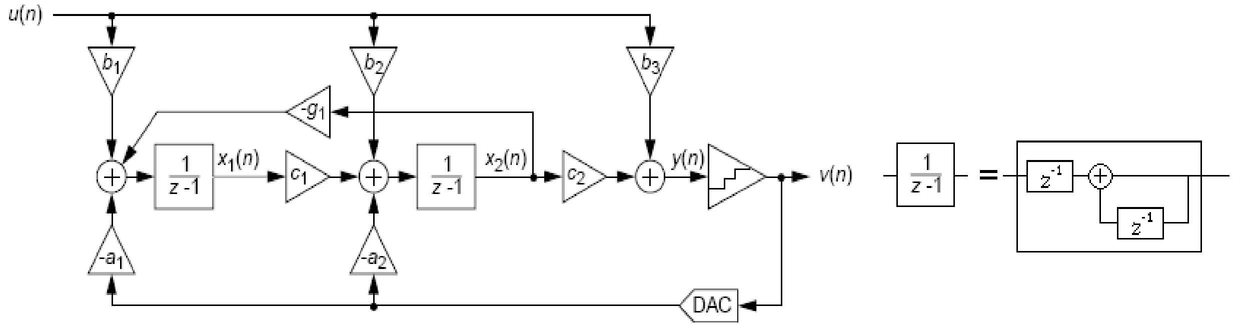


Figure 5 - A Second Order CIFB Modulator Schematic
Figure From Schreier, R.[5]

Delta sigma modulation modeling is made easier by replacing the non-linear quantizer with an additive error signal E(z). This error signal is the difference between the integrated signal and the modulated signal. From this model we can obtain the formula V(z) = [z-1 • U(z)] + [(1-z-1) • E(z)]. This equation can be written in terms of a noise transfer function (NTF) and a signal transfer function (STF). The STF of the modulator defines the frequency region which the modulator will pass from input to output. A modulator used for audio will have an STF of 1 or it will be a low pass filter surrounding the audio band. The NTF defines the region that quantization noise will exist. For an audio modulator the NTF will be a high pass filter. An effective modulator will separate the NTF and STF as far as possible to ensure quantization noise does not over run the signal band. A plot of the NTF of a first order modulator is depicted in Figure 6.

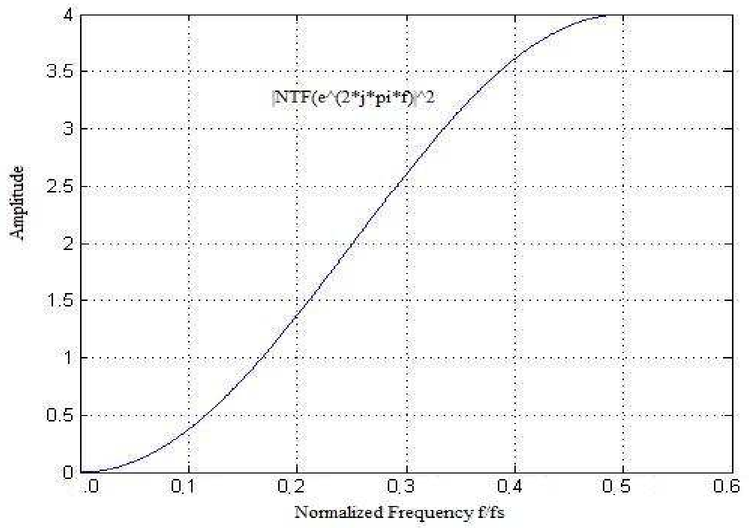


Figure 6 - The Noise Transfer Function of a Delta Sigma Modulator

If the NTF and STF of a modulator are known, the modulator can be turned into a single loop filter with two filters G and H, as shown below in Figure 7. In this diagram, G = $\frac{STF}{NTF}$ and H = $\frac{1-NTF}{STF}$, which can be confirmed by solving the output in terms of the input and the error. Both G and H filters are the same order as the order of the delta sigma modulator that they produce.

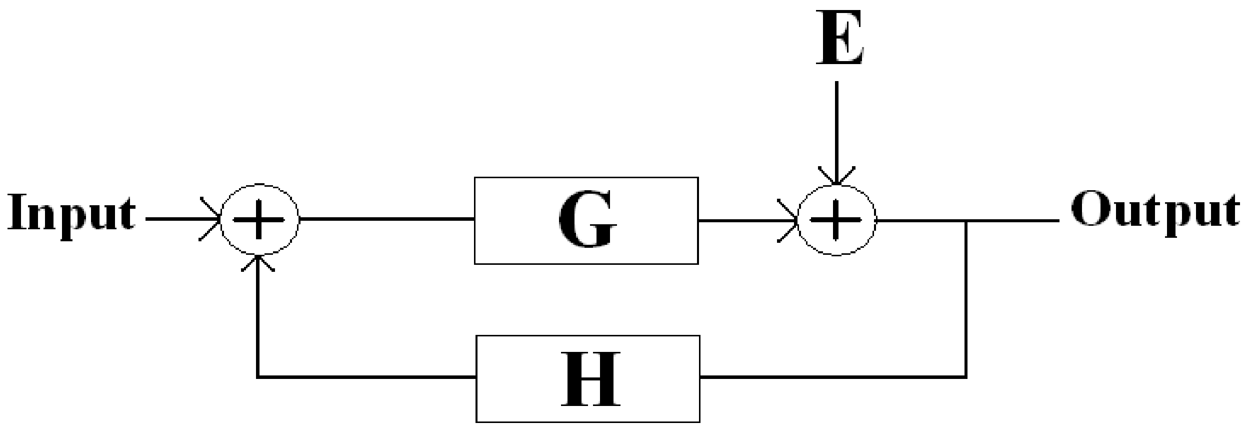


Figure 7 - Loop Filter diagram

Two parameters govern the effectiveness of a Delta Sigma modulator; those are the OSR (over sampling ratio) and the order of delta sigma modulator. In continuous time models, the order of the modulator is directly related to the number of integrators in the signal path. In a discrete time modulator, the order of the system is the order of the NTF. A higher order NTF increased the slope of the cutoff of the filter, decreasing the noise in the signal band. The over-sampling ratio (OSR) is OSR $=\frac{Fs}{2*Fb}$, where Fb is the bandwidth of the signal and Fs is the sampling frequency of the modulator. Increasing the OSR shifts the NTF out of the signal band. From figure 7 we see in a discrete time system that the noise filter is a function of the normalized frequency. When the OSR is high the signal band moves lower in the normalized frequency. The expected SNR of these systems can be estimated from the order of the system and the OSR (assuming the OSR >> 1, typically 22 to 210). In order to achieve a specific SNR, there is a trade-off between the order of the modulator and the OSR needed. Figure 8 below shows the trade-off between OSR and maximum SNR for a 2-bit Nth order delta sigma modulator. In this figure, SNR is shown as SQNR or Signal to Quantization Noise Ratio. For our purposes, these terms are synonymous.

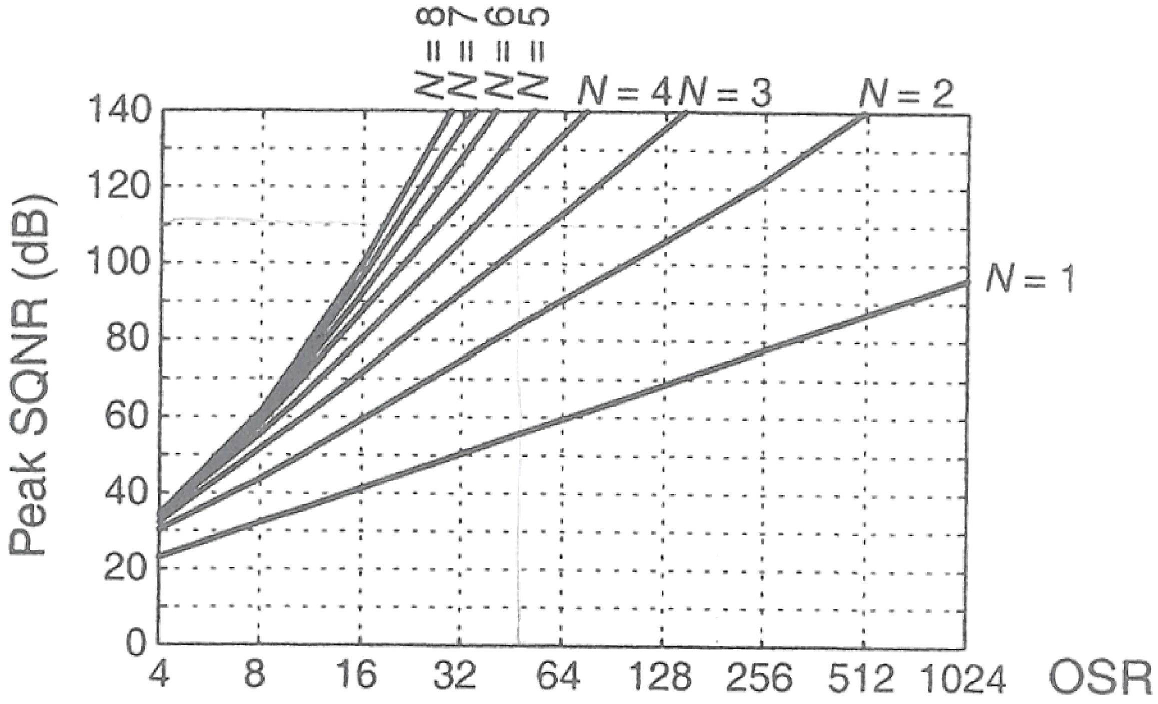


Figure 8 - OSR vs. Maximum SQNR for a 2-bit Delta Sigma Modulator
Figure from Schreier[5]

As stated earlier, delta sigma modulation can offer superior SNR because of its internal feedback and its ability to shape the noise out of the signal band. This fact makes the delta sigma modulator a good choice for an amplifier where high audio quality is required. Since the quality of the delta sigma modulator is dependent on both OSR and the order of the system being high, and a feedback loop required for the system, the stability of the modulator must be considered. Increasing the order of the modulator increases the phase delay of the system, which could potentially destabilize the modulator for some or all input amplitudes.

2.2.3 Signal to Noise Ratio

Signal to Noise Ratio is defined as shown in Equation 1. This means that the SNR of the system can only be improved by increasing the power of the desired signal or by reducing the power of the noise.

$$SNR = 10log10\left(\frac{signal\ power}{noise\ power}\right)$$

Equation 1 - Definition of SNR

In the previous sections we introduced the NTF of a modulator. We explained that increasing the order of the modulator increases the order of the NTF. Increasing the modulation frequency moves the NTF further out of the signal band. Moving the NTF out of the signal band increases the SNR of the system. An alternate way to categorize the modulator's SNR is by using its effective number of bits (ENOB). As mentioned in previous sections, the number of bits used in an analog to digital converter is directly related to its SNR. With a delta sigma modulator we are limited to a low number of output states to remain compatible with a Class D power stage. In a delta sigma modulator we can compensate for less output bits with a higher modulation frequency. By keeping the modulation frequency high we can maintain a high ENOB and therefore maintain a high SNR even with a small number of output bits.

2.3 Feedback System Stability

Having full system feedback in a Class D amplifier system has the potential to greatly decrease both noise and distortion. Like in any other feedback system, the output can be monitored and checked against the input in order to eliminate noise or distortion that is added by the system itself. One of the main challenges involved in building a system with feedback is making sure it is stable for all expected inputs. According to Dorf(1989), Feedback systems are considered stable if for any bounded input, they produce a bounded output. One way to determine system stability is by referencing the phase margin of a system. There are also many circuits which can improve the stability of a system, three of which are named phase-lead, phase-lag, and lead-lag networks.

2.3.1 Phase Margin

Phase margin is a widely used measure of system stability when working in the frequency domain. Phase margin is defined as the distance of the system transfer function's phase shift from 180º when at the system's zero dB point or the frequency at which the system's amplification factor is 1. The phase margin's positive distance from zero can determine the stability of a system. This makes sense, because a system inherently oscillates when its gain is one and its phase shift is 180º. When a system's phase margin is below zero, the system is considered unstable, because its output could approach infinity. Phase margin can be determined using Nyquist plots such as the ones explained in the previous section, but it can also be measured using Bode plots. On a system's Bode plots, the phase margin represents the frequency response's distance from -180˚ at the point where the system has zero gain. This point is known as the zero dB point.

2.3.2 Increasing System Stability

There are several approaches to making a system more stable. Many of them are rooted in changing the design characteristics of the functional blocks of the system. If this sort of change is not possible, or is not enough to make the system stable, another approach can be taken. This approach is to add another block to the feedback loop, GC(s). This function is called a compensator, because it compensates for the system's previous inadequacies.

One type of compensator is the phase-lead network. This block is made simply from a resistor which is in parallel with a resistor and capacitor and has the transfer function given below in Equation 2. In this equation, R_1 is the resistor in parallel with the parallel combination of R_2 and C.

$$G(s) = \frac{R_2}{R_2 + \{\frac{R_1\left(\frac{1}{Cs}\right)}{\left[R_1 + \left(\frac{1}{Cs}\right)\right]}\}}$$

Equation 2 - Transfer Function of Phase-Lead Network

This block also has a positive phase angle of almost 90 degrees. Since blocks in series simply add their phase responses, this block's positive phase response can directly increase the phase margin of the entire system. Another side effect of the phase lead function is that it is an amplifier for frequencies greater than the location of its own zero. This could distort the audio signal in our project if the necessary frequency of this block's zero is lower than 20 kHz.

Another type of compensator is the phase-lag network. This block is made from the same components as the phase-lead, with them arranged in a different way. For the phase-lag network, there is a resistor in parallel with a series combination of another resistor and a capacitor. The transfer function for this block is given below in Equation 3. For this equation, the R_1 is the resistor in parallel with the series combination of C and R_2.

$$G(s) = \frac{R_2 + \left(\frac{1}{Cs}\right)}{R_1 + R_2 + \left(\frac{1}{Cs}\right)}$$

Equation 3 - Transfer Function of Phase-Lag Network

This block actually has a negative phase response at a very low frequency. This should not affect the phase margin of the system, because the frequencies at which the phase-lag network's phase response is negative should be much lower than the crossover frequency of the system. Instead of directly affecting the phase margin of the system, the phase-lag network adds attenuation, moving the crossover frequency to a much smaller frequency. At this new frequency the phase margin will be greater and therefore the system will be more stable. Because this block attenuates some parts of the audio band of frequencies more than others, no analog audio signal can be passed through it without being distorted.

2.4 Digital Processing

The core of our project was the digital modulator. A digital approach implies a range of design considerations that are different or not needed in an analog approach. Questions regarding the different types of digital input standards and connectors, and quantization bits and its effects on SNR, need to be answered prior to beginning the design. More specific DSP-related issues, such as the required features of the development environment, floating point versus fixed point calculations, core processor speed, and digital output capability, are all addressed along with the previous questions in this section.

2.4.1 Digital Inputs

The consumer digital audio market is dominated by a standard know as S/PDIF (Sony/Philips Digital Interconnect Format). The standardized name for SPDIF is IE60958-3, also referred to as AES/EBU where it is known as IE60958 Type II. Type I uses a balanced 110-ohm twisted pair wire with an XLR head, this setup is used in professional audio installations where the XLR connector is commonly used for many mic level sources.

Type II comes in two formats, unbalanced and optical. The unbalanced signal is transmitted on a 75-ohm coaxial cable with an RCA head. This wire is generally utilized in consumer audio applications where the RCA connector is already commonly used for both mic and line level sources as well as many video sources. The second format of type II uses an optical cable, consisting of plastic or glass with an F05 (or trade name TOSLINK) connector. In consumer applications the F05 optical connector has been limited to transmitting digital audio.

There is no difference between the signals transmitted with either cable connector combination. Selection of either connector relies primarily on availability and consumer preference. On a more technical level, for connections longer than 6 meters or where tight bends are required, coaxial cables should be used because of the attenuation of the fiber wires is rather large and limits the effective range. However, if distance and routing is not an issue, the optical wire is superior because it is impervious to distortion and noise caused by RF interference and ground loop issues. A final deciding factor may be cost. The ubiquitous 75-ohm coaxial cable with RCA connector is given away with nearly every piece of audio equipment, while the fiber cable is rarer and therefore more expensive.

SPDIF can be used to transmit a number of different formats. The two most common formats being the 48 kHz signal used in DAT (digital audio tape) and 44.1 kHz used in CD audio. In order to support a full range of formats the SPDIF protocol has no defined data rate. The SPDIF protocol uses BMC (bi-phase mark code) which encodes the original word clock into the data stream by transmitting two bits to represent a signal data bit, allowing it to be recreated by the receiver. A 48 kHz sample rate results in a bit rate of 3.072 MHz and a clock rate of 6.144 MHz

SPDIF transmits 32 bit data streams of PCM encoded audio. The audio signal is divided into blocks, frames and sub frames. A sub-frame is a single audio sample, including some header data, channel data, parity data, etc. A frame contains a single sub-frame from every channel in the system. A block is made up of frames, the block structure is used so that additional data can be transmitted at the end of each sub-frame.

2.4.2 SNR and Quantization Bits

The core of our Class D amplifier is the modulation which is performed in the DSP. The nature of this modulation is therefore digital. Whether the input to the modulator is analog or digital, it will have to be converted, using an ADC, into a digital sample. This means that the input signal is quantized, and the sample represented using an n-bit value. Our project goals require over 100dB SNR, and it is important to understand the relationship between quantization and how it can limit our maximum attainable SNR. In this section we try to understand this relationship, and determine the minimum bits needed to represent the sample such that our SNR is not limited below 100dB.

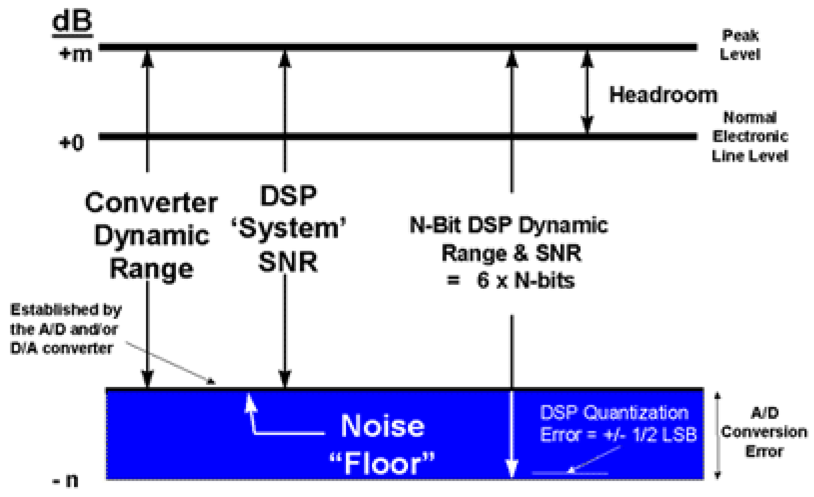
In analog terms, SNR refers to the ratio of the largest known signal to the noise present when no signal exists[1]. In digital terms, SNR and dynamic range are used interchangeably to refer to the ratio between the largest number that can be represented and the quantization error or quantization step. Higher SNR means a higher audio signal quality, as the following table indicates:

Audio Device/Application	Dynamic Range
AM Radio	48 dB
Analog Broadcast TV	60 dB
FM Radio	70 dB
Analog Cassette Player	73 dB
Video Camcorder	75 dB
ADI SoundPort Codecs	80 dB
16-bit Audio Converters	90 to 95 dB
Digital Broadcast TV	85 dB
Mini-Disk Player	90 dB
CD Player	92 to 96 dB
18-bit Audio Converters	104 dB
Digital Audio Tape (DAT)	110 dB
20-bit Audio Converters	110 dB
24-bit Audio Converters	110 to 120 dB
Analog Microphone	120 dB

Table 3 - Dynamic Range of Various Audio Devices
Figure from Tomarakos, J.[6]

[1] Another common definition for SNR is the ratio of the power of the noise of a signal to the power of the signal without the noise.

In an ADC, the difference between two consecutive binary values is known as the quantization step or quantization level. The size of the step defines the noise floor. The noise floor is the point where the audio signal cannot be distinguished from low-level white noise. These terms can be visualized as follows:



dB
+m
+0
-n
Peak Level
Headroom
Normal Electronic Line Level
Converter Dynamic Range
DSP 'System' SNR
N-Bit DSP Dynamic Range & SNR = 6 x N-bits
Established by the A/D and/or D/A converter
Noise "Floor"
DSP Quantization Error = +/- 1/2 LSB
A/D Conversion Error

Figure 9 - SNR and Noise Floor
Figure from Tomarakos, J.[6]

Simply put, the higher the number of bits, the smaller the quantization step, and therefore the higher the SNR. The formula for the SNR of an ADC, where n is the number of bits, is given as:

$$\text{SNR ADC(RMS) [dB]} = 6.02n + 1.76 \text{ dB}$$

Note that 1.76dB is based on sine wave statistics and can vary for other types of waveforms. For large enough n values, 1.76dB can be ignored and the SNR is estimated as 6n dB, or an additional 6 dB SNR for every additional bit (Tomarakos). The bottom line is that any ADC used in this amplifier must have enough bits to produce the desired signal quality (or SNR or dynamic range). The following diagram illustrates the available DSP types and their corresponding SNR's:

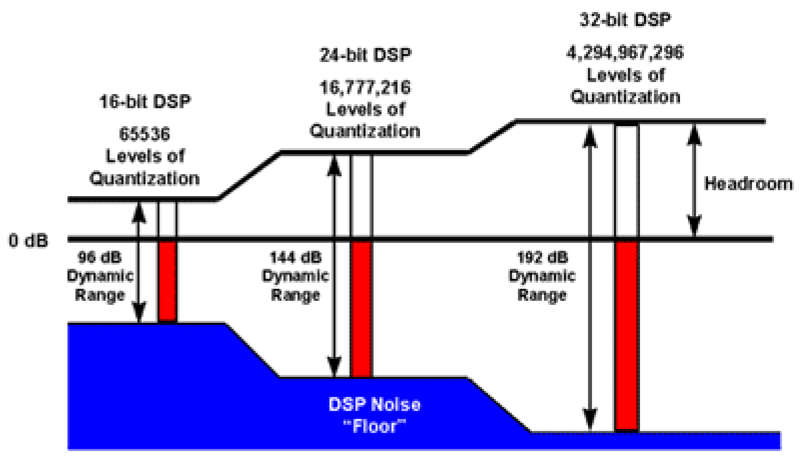


Figure 10 - Common DSP Word Formats and Corresponding SNR
Figure from Tomarakos, J.[6]

In conclusion, in order for our SNR to not be limited by any ADC used, we must choose at least a 24-bit ADC and 24bit words.

2.4.3 DSP's

Choosing the right DSP is an important step that can either hinder or facilitate our ability to perform proper analysis, testing, and debugging during the design process. This DSP must have an adequate development environment with live debugging features, and a rich interface. Other factors such as floating point vs. fixed point processors, digital inputs, digital outputs, and core processor speed must meet certain minimum criteria such that our desired project objectives are not limited by the DSP itself.

2.4.4 Development Environment

The chosen DSP must be compatible with a development environment (or IDE) that offers three main features: live debugging, statistical profiling, and optimization. The first is necessary to facilitate and speed up coding. Let us consider our approach to implementing a design on the DSP. First, we visually design the modulator in MATLAB's Simulink, using blocks for IIR filters, adders, delays, inputs, and outputs. Once the design is shown to be functional, we then translate these blocks into MATLAB code and test it once again. This offers the advantage of being able to generate a signal, pass it to our MATLAB program as an array of data, and flexibly view the results in formatted plots. After the plots show that we obtain the desired results, we then move on to the final step, which is to translate the MATLAB code into DSP code. It is in this step where live debugging becomes important. The major difference between the MATLAB program and the DSP program is that in the DSP, input is not obtained via a predefined buffer of data. Rather, it is sampled at a certain frequency, and is interrupt-driven. This means that some changes must be made to the original MATLAB code to make this possible. It also means that new problems may arise while making these changes. Without a robust and non-buggy debugging feature, a rather trivial problem may not be apparent in the code, and the results become confusing and unexpected, and can take significant time to sort out. Live debugging allows us to step through the code, and verify whether or not each step produces the expected result. When the variables differ from what is expected, we can easily determine, based on the break-point of the program, what may have caused the error. As a note, the ability to easily watch variables in real time is also a desirable IDE feature.

As aforementioned, the DSP obtains samples at a certain frequency. Whenever a new sample is ready, an interrupt is generated, and triggers the predefined interrupt service routine (ISR). This offers the ability to process a signal in real time, given that the code in the ISR completes its function before the next sample arrives. Indeed, this poses a strict time limit to the code in the ISR, which we must take into account when coding our modulator. Without some kind of statistical profiling available in the IDE or provided with the IDE's libraries, measuring the time taken to execute the code in our ISR is a difficult task which involves approximating the number of CPU instructions our code will generate to calculate the total time given the time taken per instruction. When different types of instructions vary in execution time, and if the code is written in c, the task is even more complex. The most efficient way to measure execution time is to start a timer prior to the first instruction in the ISR, and stop it after the last instruction, then convert the timer result and display it in seconds. The IDE we used did offer such a prebuilt timing function, which was helpful in indicating whether or not the code took no longer than the allowed time. Statistical profiling, however, also involves the ability to look at the data being read or generated by the DSP in a visual way. Buffering a time-varying variable (such as the input), and then displaying the buffered data in a plot within the IDE can save us the hassle of having to export the data and plot it in a third-party software. The ability to perform FFT and other analysis functions within the IDE is also a plus

The last of the three pillars of IDE requirements is optimization. Given that we lack the time or expertise to write a fully pipelined and optimized assembly code to perform our modulation function, our best option is to do it in C, which, needless to say, requires an IDE that can compile C code. However, C code is not always compiled in an efficient way. In fact, previous experience has shown that a non-optimized C code can take three times as many cycles to run as a sub-optimized assembly code would. After optimization, however, the C code became twice as fast as the assembly code, indicating a potential six-fold gain in efficiency post-optimization. The IDE's ability to provide optimization for C code is therefore important because execution time is of the essence. Having the tools to measure the execution time per interrupt, and ways to increase processing speed, the question then becomes what is the maximum execution time per interrupt that is allowed by the DSP, how do we calculate it, and what limitations does it imply?

2.4.5 Core Processor Speed

It is easy and obvious to say that the faster the CPU is, the easier it is for us to implement higher order modulators on the DSP. However, it takes some analysis to determine a reasonable constraint for the minimum core CPU speed required. Consider the following diagram:

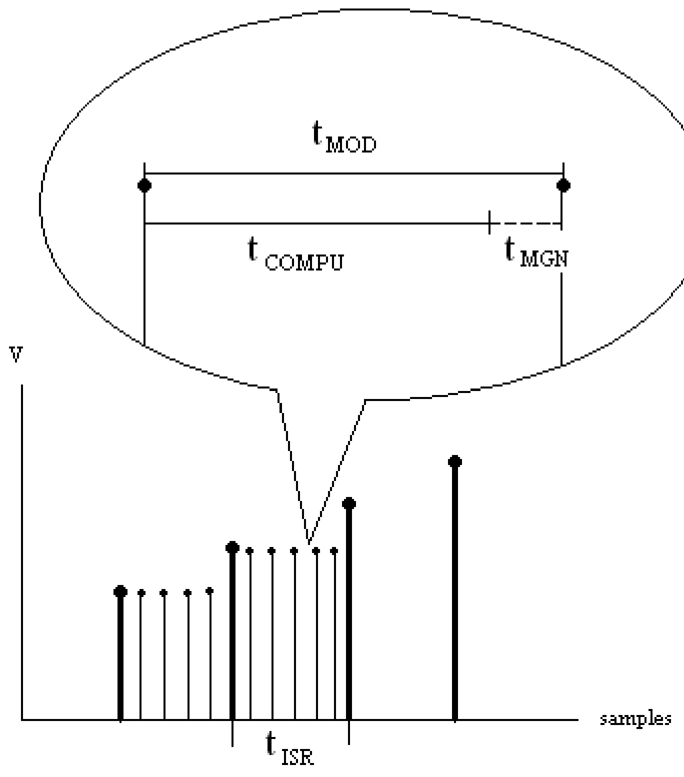


Figure 11 - DSP Sampling and Processing Timing

The bold lines indicate the samples obtained from the DSP's input, which are separated by t_{ISR}, or one over the sampling frequency. Moore's law tells us that a delta sigma modulator must output at a frequency greater than or equal to 60 times the highest frequency. In our case, the highest frequency in the audio band is 20 kHz, so 60 * 20 kHz = 1.2 MHz. The samples between the bold lines represent the samples that will be output from the DSP, and these must be output at a frequency greater than or equal to the modulation frequency (1.2 MHz). This gives us $t_{MOD} = 1/f_{MOD}$ amount of time to complete a full modulation cycle. If we let t_{COMPU} represent the time taken to complete the computation of a full modulation cycle, t_{COMPU} must be smaller than t_{MOD} to allow for a small margin of time, t_{MGN}, which will be used to wait for the next computation to begin. This defines our major constraint, which is broken down as follows:

$$t_{COMPU} < t_{MOD}$$

$$t_{COMPU} = n * t_{CPU}$$

$$t_{CPU} = \frac{1}{f_{CPU}}$$

$$\frac{n}{f_{CPU}} < t_{MOD}$$

$$\frac{n}{f_{CPU}} < \frac{1}{f_{MOD}}$$

$$\boldsymbol{f_{CPU} > n * f_{MOD}}$$

The frequency of the core CPU must therefore be greater than the number of instructions per ISR (*n*) times the modulation frequency. For f_{MOD} = 1.2 MHz, and an estimate of 300 instructions per ISR, our minimum core clock frequency must be greater than 360 Mhz.

2.4.6 Floating Point vs. Fixed Point

Knowing that we are constrained in time, and that we want to minimize the number of instructions per clock cycle, while not compromising sample integrity, we must face the question of whether to use fixed point or floating point calculations in our DSP. In an n-bit processor, fixed point numbers can be represented as signed or unsigned integers or fractionals. A signed integer ranges from 0 to 2^n , unsigned ranges from -2^{n-1} to 2^{n-1} and all values within these ranges are equally spaced. Fractionals are normalized to range between -1 and 1 (for signed) or 0 and 1 (unsigned), which is done by dividing the integer value by 2^{n-1} (for signed). Floating point numbers, on the other hand, are represented as scientific numbers (IEEE standard), and for 32bit floats, the largest and smallest numbers are $\pm 3.4\times 10^{38}$ and $\pm 1.2\times 10^{-38}$ respectively. Floats offer the advantage of being able to represent much larger and much smaller numbers than fixed point, by compromising the equality of distribution of numbers within that range. That is, large gaps separate two consecutive large numbers, and small gaps separate two consecutive small numbers.

In many DSP's, fixed point is known to be more efficient than floating point because it involves less operations and usually less bits. However, SHARC processors are designed to be equally efficient for both fixed point and floating point calculations. Setting that note aside, there is a major disadvantage in using fixed point numbers: the ease at which overflow can occur. If a calculated number turns out to be greater than the largest possible fixed point value, then we now have more bits than can be stored in the register containing the fixed point value. This produces unpredictable results, and is usually handled by pre-scaling the operands used to calculate the result such that the result does not overflow. Fixed point calculations, on the other hand, can store very large numbers while still being able to represent very small numbers. In the scenario where we are working with an input that ranges from -1.0 to 1.0, calculations can yield our desired range with high precision, while allowing results to be much greater than our range. Considering the fact that no efficiency is lost when using floating point in a SHARC processor, one can conclude that floating point is the best choice if we do use such a processor.

2.4.7 Digital Outputs

Another important consideration is the ability to output a high frequency PDM signal from our DSP with minimum distortion. We also need at least two digital outputs in order to produce a tri-state signal. These outputs must be able to switch at frequency greater than or equal to the minimum modulation frequency, or 1.2 MHz. This means the rise time or fall time of the switching, plus the response time (the time taken between the instruction to switch and the actual start of the switch) must both be smaller than or equal to $1/f_{MOD}$. The following diagram illustrates this concept:

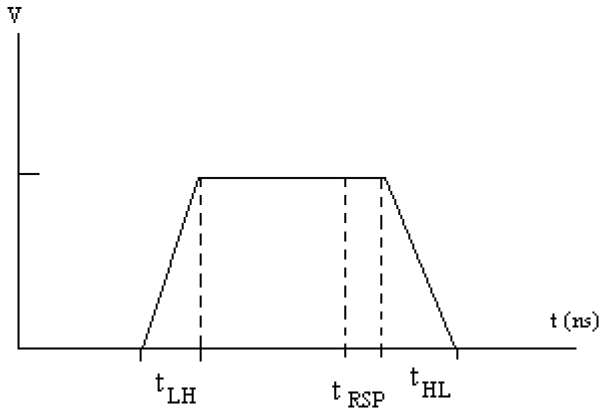


Figure 12 - Digital Output Timing Diagram

t_{RSP} : response time

$t_{LH:}$ Transition time form low to high

t_{HL} : Transition time from high to low

Every switch in state is preceded by a response time t_{RSP} which varies depending on the DSP's core speed and architecture. In short, the DSP must have 2 digital outputs, where for each of these, $t_{LH} + t_{RSP}$ and $t_{HL} + t_{RSP}$ are less than or equal to t_{MOD}.

2.5 Power

The power stage of a Class D amplifier accepts a modulated signal, amplifies it and delivers it to the filter and speaker. Because a Class D power stage is amplifying a digital signal it need only provide full power or no power. This means the transistors used are either on or off, and do not have to operate in their inefficient linear regions. The two most common configurations for a Class D power stage are H-bridge and half bridge.

In a half bridge configuration there are two transistors. One transistor is attached to the positive rail and the other is attached to the negative rail. The load (speaker) has one lead connected between these transistors and the other lead is attached to a bias voltage equal to half the difference between the power rails. Only one transistor can be on at a time, otherwise the voltage rails would short together. Enabling one of the transistors supplies the load with a positive voltage, enabling the other provides a negative voltage. Several issues exist with the half bridge configuration. This power stage is only capable of two states: positive and negative. At idle, the load will be biased above ground. Also the maximum output voltage can only be half the power supply voltage.

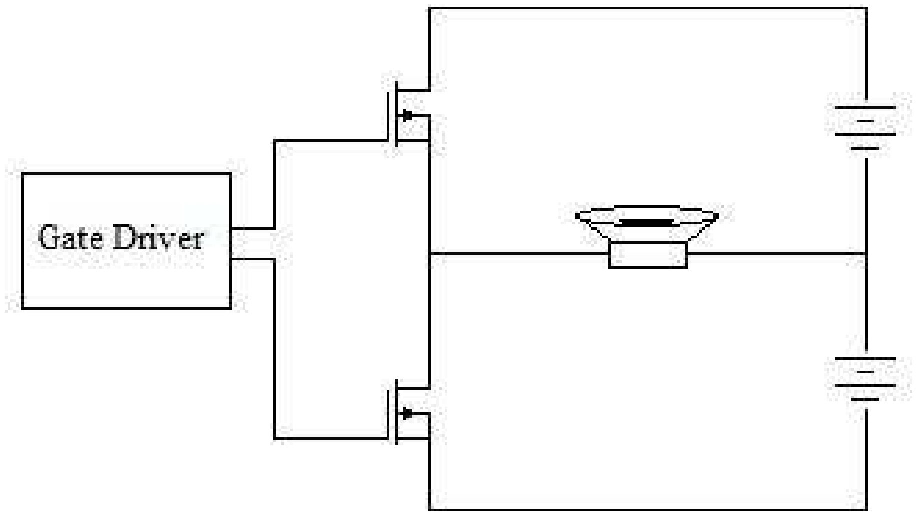


Figure 13 - Half Bridge Power Stage

In an H bridge configuration there are four transistors. Two transistors are placed in series between the upper and lower power rails, and another pair of transistors are also placed in the same configuration. The load is attached between the transistors of these pairs. This configuration is capable of three states: positive, negative, and neutral. This is accomplished by changing the polarity of the speaker leads to create the positive and negative voltage swings, and also by grounding both sides of the load to create a zero state. Because this configuration only changes the polarity of the speakers connection, it is capable of outputting the full power supply voltage to the speaker, as opposed to half the supply as in the half bridge configuration.

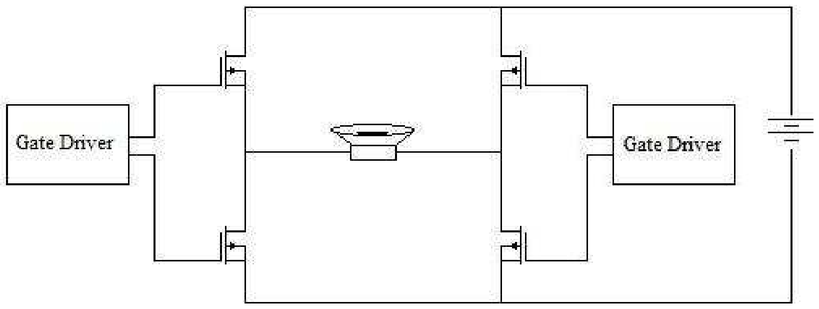


Figure 14 - Full Bridge Power Stage

The Class D MQP team from 2008 designed and built a very efficient H bridge power stage. In their testing it achieved over 95% efficiency. Because the goal of this year's MQP is the addition of digital inputs, digitally controlled modulation, and feedback control to the Class D amplifier and because the 2008 team was so successful with their power stage, it was decided that their power stage design would be reused.

2.5.1 Noise, Distortion, and Filtering

The goal of a Class D amplifier is to convert a continuous time waveform into a series of binary pulses so they may be more efficiently amplified. This conversion adds unwanted harmonics as well as other noises to the output. Some of these are audible, and therefore undesired for noise quality reasons. Other noises are inaudible, but still undesired for efficiency reasons.

A Class D modulator runs at speeds significantly higher than the highest desired output frequency, as dictated by the over sampling ratio. The noise introduced by the modulator is intentionally "shaped" out of the audio band. Despite the audio quality benefits of this "shaping", if the constant switching of the modulation stage was allowed to propagate to the speaker the overall efficiency of the system would be decreased. Although the noise added by the modulation stage is in audible, it still requires power to push the speaker cone. To increase the efficiency of the system, and reduce the possibility of cone damage because of high frequency oscillation, the output is filtered to remove this high frequency noise.

Not all noise added by the modulator will be outside of the audio band. Low fundamental frequencies will result in odd order harmonics that may lie within the audio band, increasing THD. In band noise cannot be filtered at the output.

Low pass filters can be implemented either actively or passively. Active filters exhibit exceptional frequency roll offs, and have the ability to have extremely minimal pass band rippling. Active filters do not rely on non-ideal components like inductors and capacitors, whose non-linearity will distort the filter output.

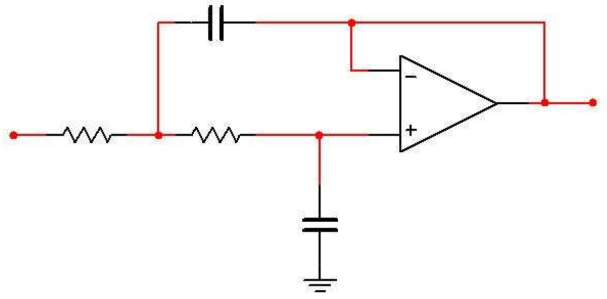


Figure 15 - Example Low Pass Active Filter

Unfortunately the basic design of an active filter makes it impractical for Class D amplifiers. Active filters are based on amplifiers themselves. Class D amplifiers are efficient because of their switching power supplies, the addition of a linear amplifier would defeat the purposes of making a high efficiency amplifier.

A passive filter is a more practical choice for a Class D amplifier. A passive filter can, at best, have unity gain. A passive Butterworth filter has a very flat pass-band, although its frequency cutoff is not as sharp as some other filter configurations. Passive filters are built with non-linear components, producing output distortion. Last year's team created a passive Butterworth output filter that performed as required, and this design was reused for this year. The following figure shows the balanced low pass filter configuration and component values used by last year's team. The component values used result in a 30 kHz cut off.

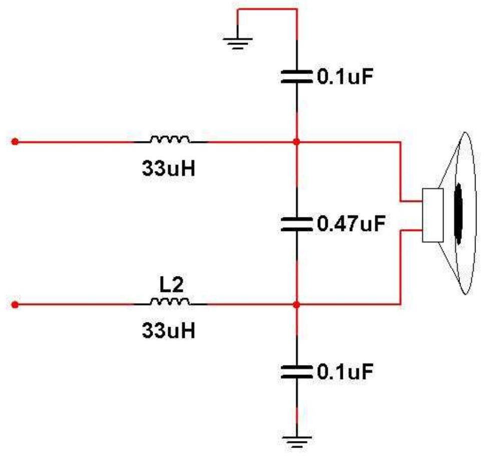


Figure 16 - 2008 Class D Audio MQP Filter Design

3 Simulation, Design and Parameter Synthesis

The design of our modulator and feedback systems were completed as iterative simulation processes. This allowed us to constantly edit major portions of design until the day we began testing. This also allows for future work to easily improve on the progress we have made, including continuing our research on full system feedback. We were able to take this approach to design because the modulator and feedback systems were implemented with software on a DSP.

The PCB design which was created with the intention of allowing our system to have feedback was only designed once. While much of this PCB design was recycled from last year's award winning project, only the newly designed portion of our PCB functioned correctly. In this chapter, the design of the modulator in both MATLAB and a digital signals processor will be discussed alongside the feedback design created in Simulink and the hardware PCB design.

3.1 Modulator

In order to design a modulator fit for a DSP, several different models were built in MATLAB before finally porting the design onto a digital signals processor. While researching and defining the general structure of the modulator, models were built in Simulink, MATLAB's graphical model based design simulator. There were two major modulator structures that were built and simulated; the cascaded integrator model and the loop filter model. The loop filter design was eventually chosen because of its portability to a DSP.

3.1.1 Cascaded Integrator Form

There are several different delta sigma modulator structures that can be used to realize any given noise and system transfer function pair. One such model is called the cascade of integrators, feedback (CIFB) structure. This modulator topology is examined in detail in section 2.2.2, the delta-sigma modulation section of the background chapter. The second order form of the modulator structure is also shown in Figure 17 below for convenience.

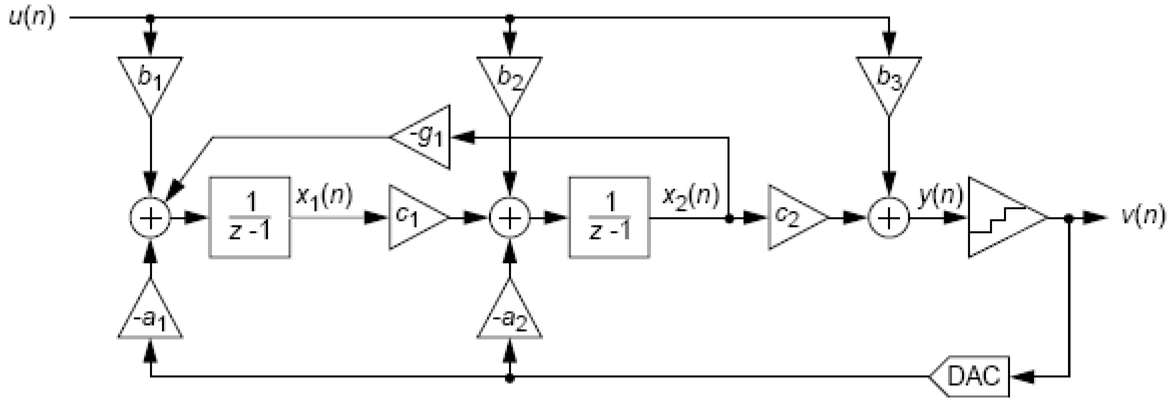


Figure 17 - A Second Order CIFB Modulator Schematic
Figure from Schreier[5]

In order to determine the value of all the constants to be used in the simulation, the MATLAB code in Appendix A was used. Running this code for a second order modulator with an OSR of 50 and plugging the resulting coefficients into the general CIFB structure resulted in the Simulink model shown below in Figure 18. Notice that the discrete time integrators in Figure 18 do not differ from the integrators in the general structure; they are simply represented with z-1 instead of z. Also notice that the variable g1 is given as zero by the MATLAB code in the appendix, so that portion of the modulator is omitted. There is a two pole Butterworth filter on the output of the modulator in order to ensure that the input sine wave can be reproduced.

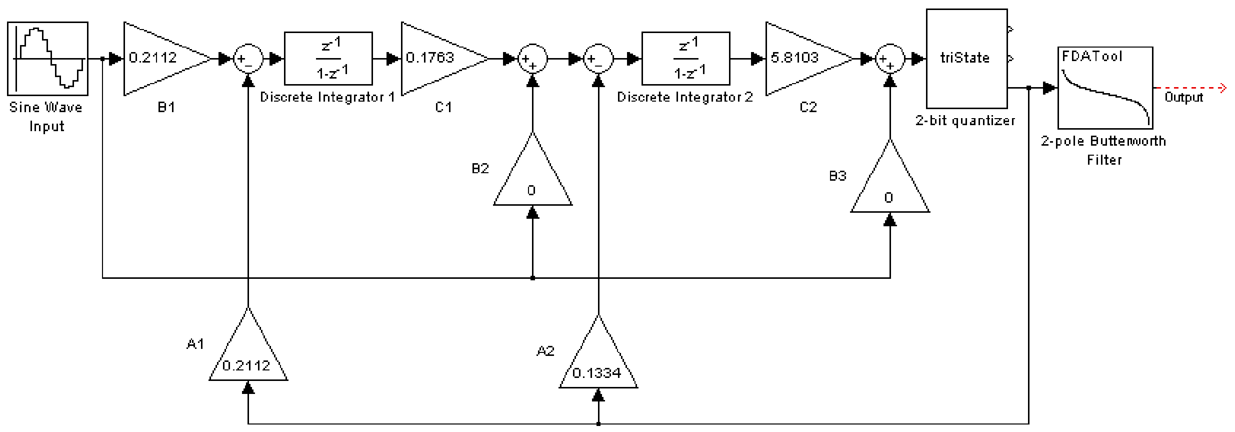


Figure 18 - Simulink Model of 2nd Order CIFB Delta Sigma Modulator

3.1.2 Loop Filter Form

In the background section 2.2.2 of this report, it was shown that a loop filter could represent any modulator topology. This modulator form has a much simpler appearance and is easily represented by C code in a DSP. The reason this form can be easily implemented in a DSP is the fact that it consists of only IIR filters. The process of actually implementing an IIR filter in a DSP is discussed in section 3.3.2 IIR Filter Implementation of this report. The general loop filter form is shown below in Figure 19 for convenience.

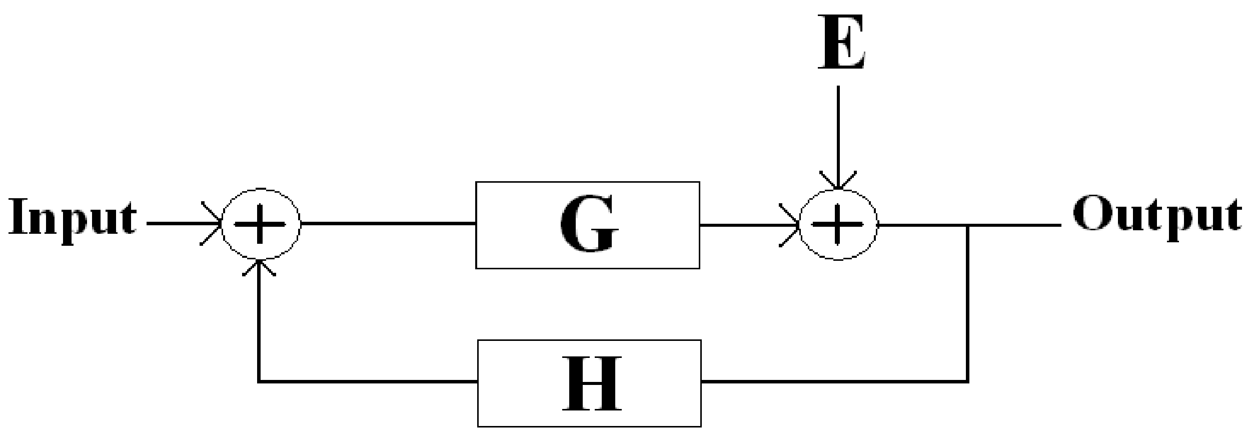
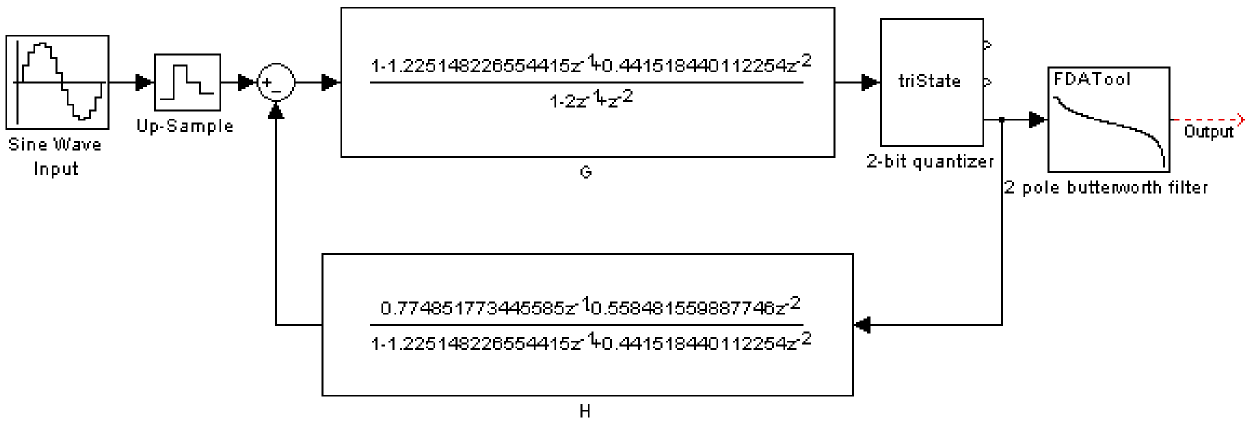


Figure 19 - Loop Filter diagram

In order to convert an NTF and STF pair into a general loop filter form, the MATLAB code in Appendix B was created. This code takes an NTF created by the delta sigma toolbox for a certain order modulator, and creates the two loop filter blocks, G and H, while assuming an STF of 1. The Simulink model shown in Figure 20 was created after running the script in appendix B and placing the resulting G and H blocks into the general loop filter model.


Sine Wave Input
Up-Sample
1-1.225148226554415z^-1+0.441518440112254z^-2
1-2z^-1+z^-2
G
triState
2-bit quantizer
FDATool
Output
2 pole butterworth filter
0.774851773445585z^-1-0.558481559887746z^-2
1-1.225148226554415z^-1+0.441518440112254z^-2
H
Figure 20 - Second order Loop Filter Simulink Model

In this model, the error added to the output is replaced by a two bit quantizer. Once again, a 2 pole Butterworth filter is attached to the output in order to test if the input sine wave can be re-created from the output of the modulator. Only the second order model is shown here, but models for orders 1-5 can be seen in Appendix C.

3.2 Feedback

Feedback can theoretically reduce both the noise and distortion of almost any system. Feedback can also make the same system unstable if it is implemented without great attention to delays and phase shifts within the system. When conceptualizing our amplifier with full system feedback, there are two blocks that attribute significant phase shift. The first is the two pole

Butterworth filter used to low-pass our output just before placing it across the speaker. This filter adds more than 80º phase shift to the system before the end of the audio band.

This second source of phase shift comes from the delay in the analog to digital converter going into the DSP that executes the modulation. The output of the amplifier is fed back as an analog signal, but must be passed into the DSP in order for it to be used in future modulation iterations. Therefore, it must be passed through an ADC, all of which have a relatively long delay called a group delay. For the ADC we had available, the time between when the analog signal first enters the ADC and when a digital representation of that voltage is available at the output is 460µs. Because of the phase shift induced by a delay is 360*f*d, even a 1 KHz signal would have 165.6º phase shift from the group delay alone.

Since the audio band contains information up to 20 kHz, phase shift induced by the group delay would be up to 3312º. This would make the system wildly unstable if feedback were implemented this way. There are several things that can be done to avoid these stability problems, while still enjoying the benefits of full system feedback.

3.2.1 Stability Remedies

While designing implementations of feedback for our amplifier, two techniques were used to combat the crippling phase shift that was added to higher frequencies by the ADC's group delay. Phase shift is directly related to delay by the following relationship: phase shift = 360*f*d. By feeding back only the lower frequency band of the output, the phase shift can be reduced to a minimal and stable value. In order to only feedback lower frequencies, a low pass filter can be added to the feedback network. Although this compromise would greatly increase stability, it would also remove any of the benefits of feedback for the higher frequency bands.

In order to actually implement this idea into our Simulink models, it was decided that two modulators could be used instead of one. The first modulator would be exactly the same as the original, but the second modulator's H-block would take its feedback from an attenuated, low pass filtered, and delayed version of the amplifier's output. A Simulink model of the system that was described is shown below in Figure 21. Notice that in Figure 21, the first modulator is represented by G1 and H1, while the second modulator is represented by G2 and H2. In this model, the LPF used to eliminate higher frequencies from the feedback path is created by the 1 kHz feedback filter and the inverted version of that filter. Having these two filters where they are adds a 1 kHz LPF to the signal transfer function of the second modulator.

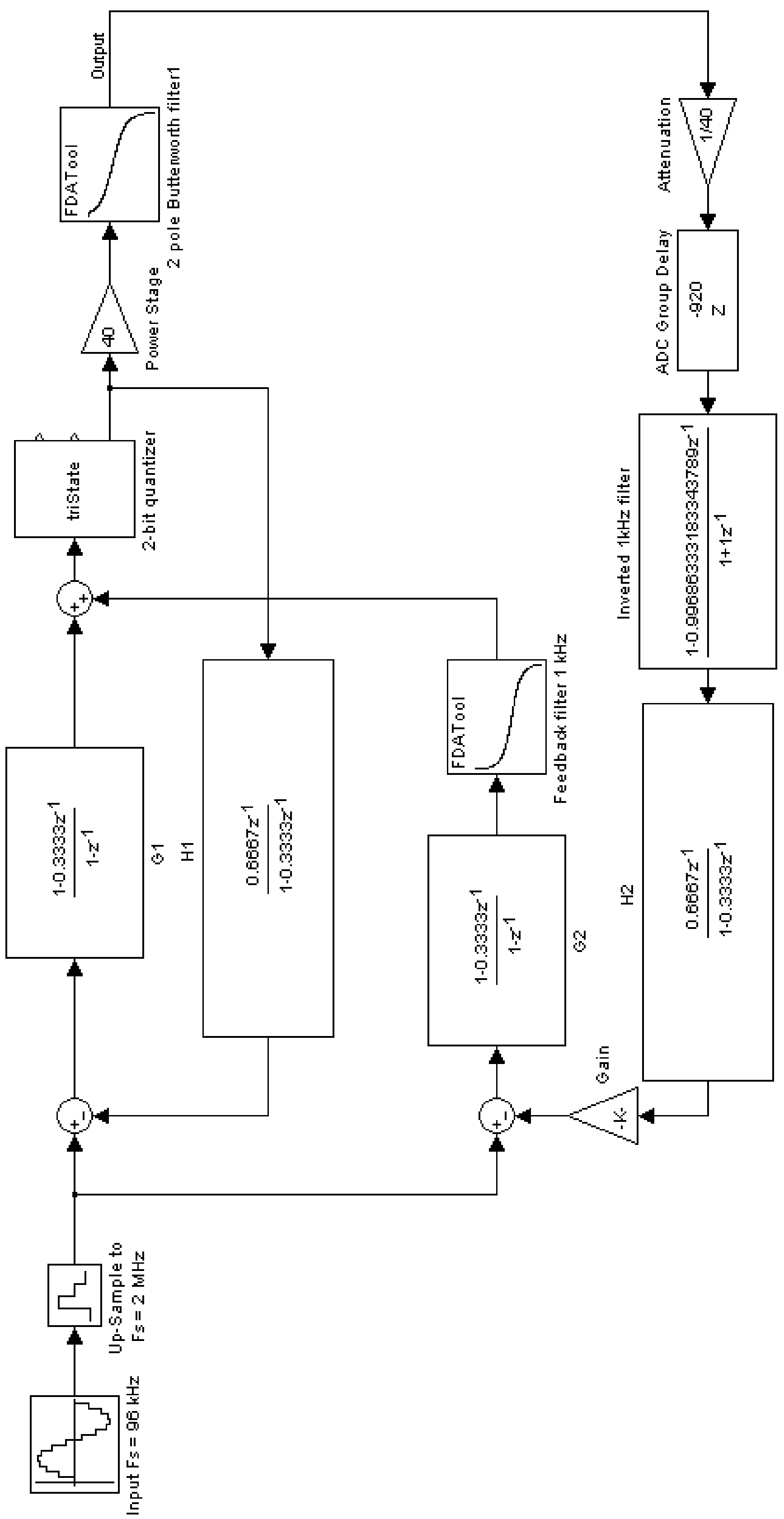

Input Fs = 96 kHz
Up-Sample to Fs = 2 MHz
$\frac{1-0.3333z^{-1}}{1-z^{-1}}$
G1
H1
$\frac{0.6667z^{-1}}{1-0.3333z^{-1}}$
2-bit quantizer
triState
40
Power Stage
FDATool
2 pole Butterworth filter1
Output
$\frac{1-0.3333z^{-1}}{1-z^{-1}}$
G2
FDATool
Feedback filter 1 kHz
Gain
-K-
H2
$\frac{0.6667z^{-1}}{1-0.3333z^{-1}}$
Inverted 1kHz filter
$\frac{1-0.996863331833343789z^{-1}}{1+1z^{-1}}$
ADC Group Delay
Z^{-920}
Attenuation
1/40

Figure 21 - Second Order Modulator with Full System Feedback Simulink Model

3.2.2 Complications

After building the Simulink model shown in Figure 21, it became clear that optimizing a system with feedback required a huge amount of research on the topic. After building the model, several parameters, such as the gain labeled "gain" and the coefficients of the second modulator were fairly arbitrarily chosen in order to test the prototyped feedback model. The system produced a completely stable output with the same shape of the input. The model produced similar quality outputs for inputs both above and below the 1 kHz feedback cutoff frequency.

The major concern with this model was that it failed to produce a better SNR for inputs below 1 kHz when compared to the SNR with higher frequency inputs. Also, the SNR of the system for any given input frequency was approximately equal to or less than the SNR produced by a comparable Simulink model without feedback. Several small variations of this model were made, but none of them produced more favorable results. This showed that the feedback system in its current state was not advantageous for any reason. Because this was determined fairly late in the project, there was no time to begin a new round of research on this type of feedback. The decision was then made to eliminate feedback from our final design.

If full system feedback were to be implemented in the future, one of two things would need to be done. The first option would be finding an analog to digital converter with a shorter delay time than the one used this year. The delay time of the ADC we used was the reason such a complicated system was needed to implement feedback. If an exceptionally fast ADC was found, the feedback could be implemented directly, without the need for a low pass filter in the feedback path or a second modulator. For such a model to be stable, the group delay of the ADC in the feedback path would have to be 25 μs at absolute maximum. Even that short group delay would induce a phase shift of 180° at a signal frequency of 20 kHz. As was discussed in the stability section of the background, section 2.3.1, a phase shift greater than 120° is undesirable, because it makes the system marginally unstable.

To make our current feedback model produce better audio quality results, the transfer function of the whole feedback system would have to be derived, examined, researched, and improved. For our project, we only had time to examine each of the feedback system's individual modulator blocks, which made it difficult to assess the reason it the model was performing poorly. More research on the topic may even reveal a better feedback model or a better way to implement the one our team attempted.

3.3 DSP Oriented Simulation

With a clear visual picture of the overall modulator system in MATLAB's Simulink, the next step is to convert this model into an n-order MATLAB function with the dual purpose of simulating the modulator similarly to the way the DSP will run its modulation code, and creating a base program that will make it easier for us to transition to DSP code. Simulating the DSP code in MATLAB will allow us to take advantage of MATLAB's powerful plotting and analysis tools to evaluate and predict results in a practical way. This section describes how we implement the modulation code in MATLAB, delves into IIR filter implementation, and shows how we took advantage of MATLAB to optimize coefficients and predict SNR results.

3.3.1 Software Flow Diagram

The goal of our program is to generate a test signal, given a user defined input frequency and duration, modulate the signal, given a sampling frequency and modulator order, and return an output signal which can then be plotted and analyzed. The software flow diagram in Figure 22 details the overall operation of the modulation function DSPsim(). The MATLAB code for this function, and all its sub-functions, is provided in appendix D.

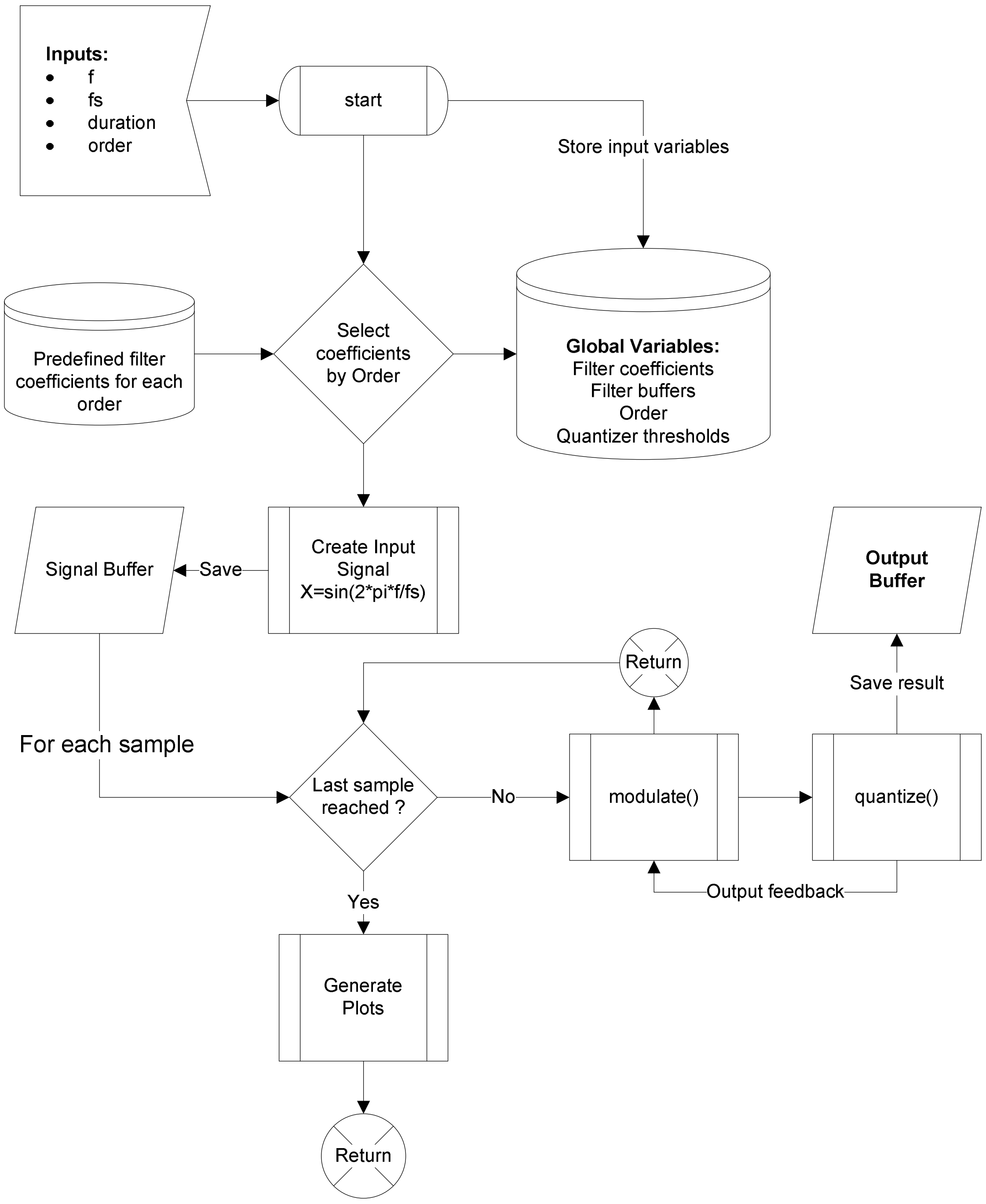


Figure 22 - DSPsim() Function

The user calls the DSPsim() function, providing the inputs shown in Figure 22 above. The function has the filter coefficients for all five modulator orders predefined, and begins by using a switch that selects the coefficients which correspond to the given order. It then creates a test signal that has the duration and frequency defined by the user. This signal is simply a sine wave. Then, the modulate() function is applied for each sample of that signal. This function contains the two IIR filters used for the modulation, uses the filter coefficients which have been previously selected, and uses the quantize() function internally (hence the feedback path from quantize() to modulate()). The modulate() and quantize() functions are illustrated in Figure 23 and Figure 24 respectively.

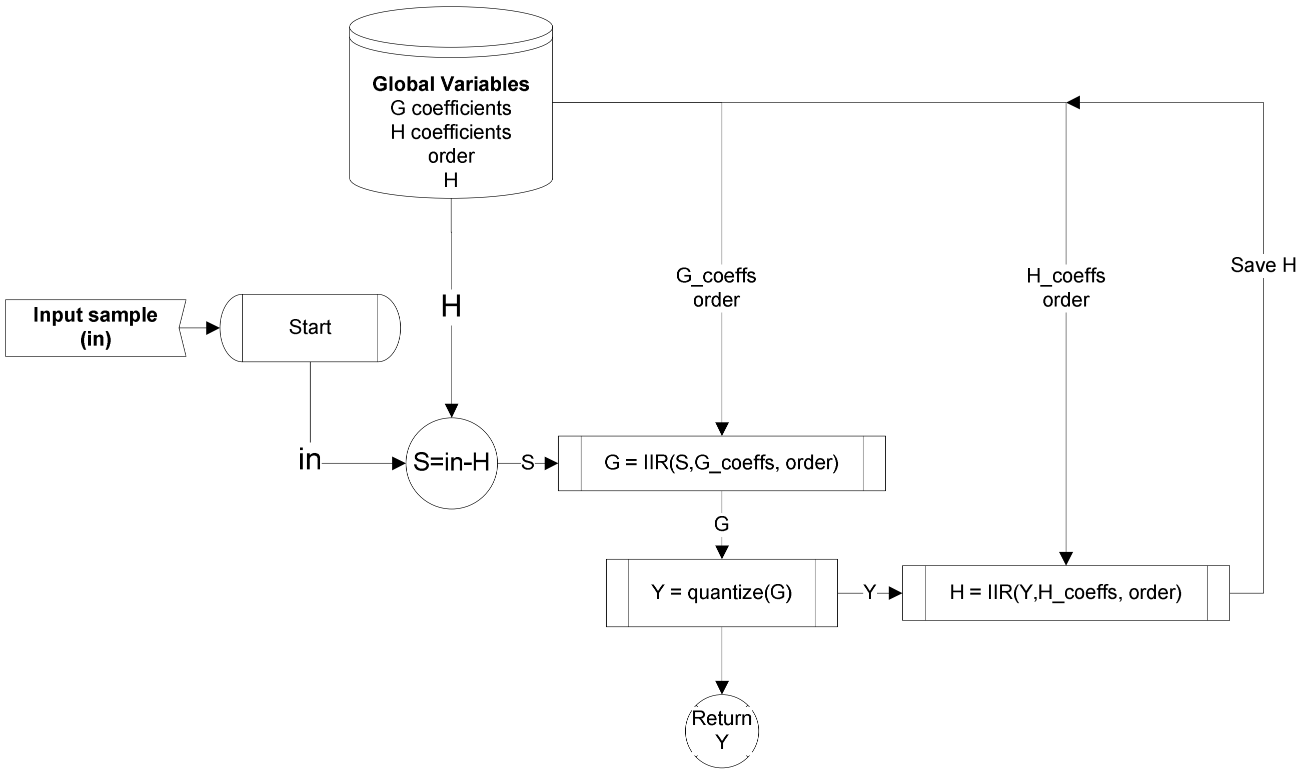


Figure 23 - modulate() Function Block Diagram

The modulate() function has the same form as that of the Simulink models. It uses two IIR filters, called G and H, to perform the transfer function defined by the coefficients. Note that these coefficients are passed as global variables. The output of the H filter is stored as a global variable every time this function is called, meaning that the H value used to calculate S is that of the previous modulation. For clarity, we can say that S[n] = input[n] - H[n-1]. The output Y is obtained by quantizing the result of the G filter, and then passed to the H filter. Once the H filter completes its calculations, the new H is saved, and Y is returned, ending the function.

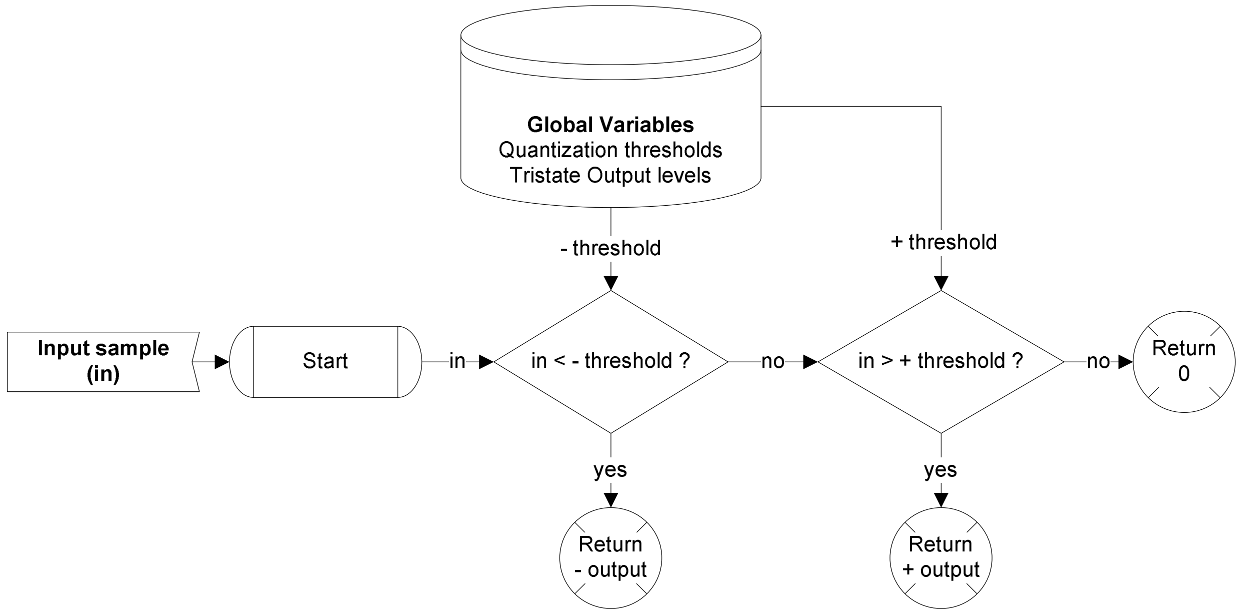


Figure 24 - quantize() Function Block Diagram

The quantization step is rather trivial. Threshold and output levels are defined in the DSPsim() function and stored as global variables, and are illustrated in Figure 25 below:

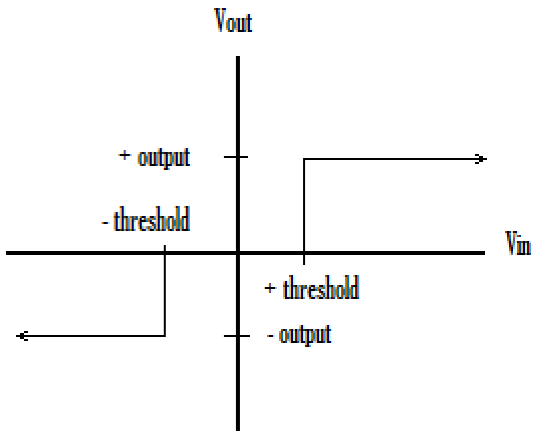


Figure 25 - Tri-state Quantizer Levels

3.3.2 IIR Filter Implementation

In many ways, the IIR function is the most important one. Its implementation and efficiency will eventually define the maximum speed at which our DSP can modulate. While being efficient, it must also be generalized to work for any order, so that switching between orders becomes a simple matter of choosing the right coefficients and order. This will in turn make testing for different orders easier. The IIR filter implementation we chose was the direct form II structure, shown in Figure 26. It offers the advantage of being both easy to implement, and efficient.

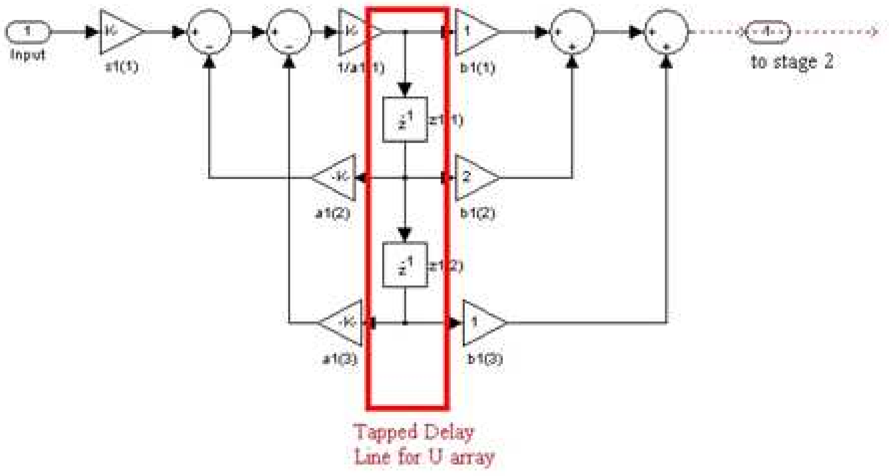


Figure 26 - Direct Form II IIR Filter Structure (Source: MATLAB Help)

The red box shown in the figure above represents the buffer of data which contains the output of *n* previous filter calculations. This buffer is then "tapped" with *a* coefficients to produce the new value to store in the buffer, and then tapped with *b* coefficients to produce the output. Compared to the direct form I implementation, only a single buffer is used instead of two, which is why it is more efficient. Practically, the tapped delay line, or filter buffer, must be "circulated" every iteration of the filter. This means that all values in the buffer need to be shifted forward to open a spot in which the new *u* will be stored. The process of copying data from one array position to another takes a considerable amount of clock cycles, and this must be done *n* times per iteration, according to the order of the filter. Therefore, minimizing buffer circulations is one way to improve efficiency. The IIR structure in Figure 26 can be distilled into the following two equations for *u* (the filter buffer/delay line) and *y* (the filter output).

$$u[n] = g * x[n] - a[1] * u[n-1] - a[2] * u[n-2]$$

Equation 4: Updating the current buffer line *u* (*a* taps)

$$y[n] = b[0] * u[n] + b[1] * u[n-1] + b[2] * u[n-2]$$

Equation 5: Output : Calculating the output *y* (*b* taps)

The next step is coding the IIR filter. This process involves circulating the *u* buffer, then applying Equation 4 and Equation 5 from above to calculate the new *u*, and return the resulting *y*. Figure 27 illustrates the IIR function we created.

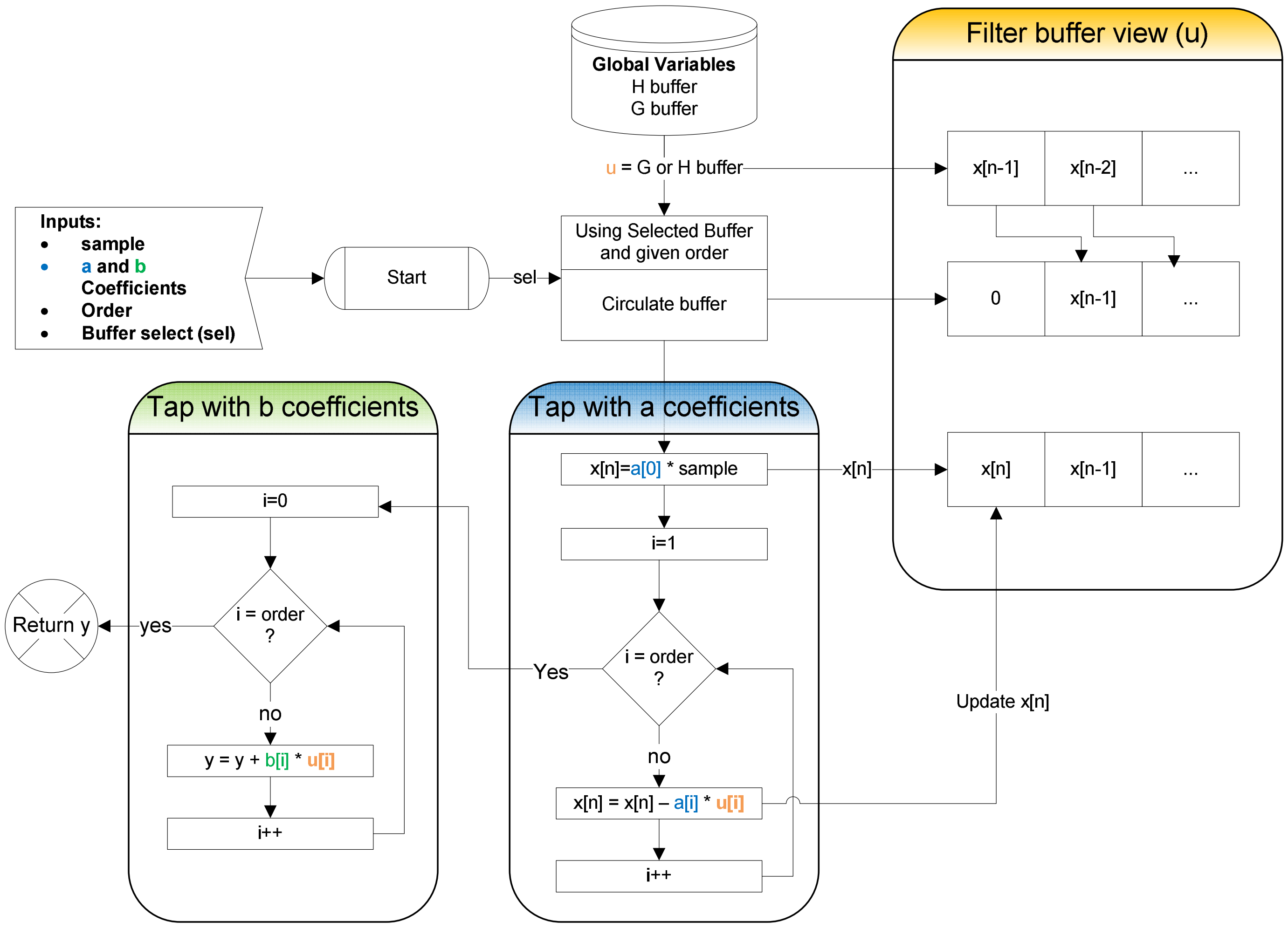


Figure 27 - Software Flow Diagram for an N-Order IIR Filter Function

The dspSim() was tested and functional, and the following is a sample of plots it generated using a 3rd order modulator, a 2kHz sine wave test signal, and a modulation frequency of 2MHz:

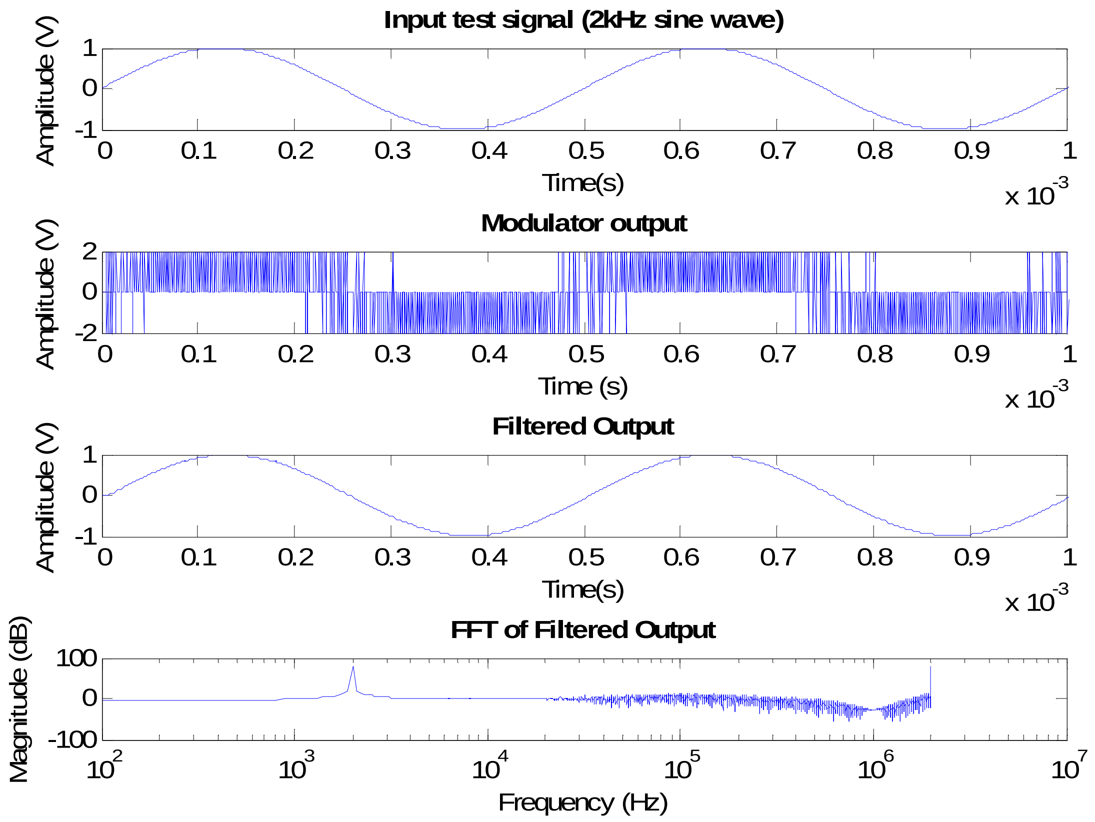


Figure 28 - dspSim() Output Sample

3.3.3 Coefficient Optimization

With a tested functional modulator simulator, we were then able to optimize the modulator before moving on to the implementation on the DSP. In order to find a better set of coefficients for each order, we took advantage of MATLAB's fminsearch() function to maximize SNR by tweaking the coefficients as a parameter. Fminsearch uses an algorithm known as the Nelder-Mead simplex method, which tweaks the function's input parameters little by little until the value returned by the function is minimized. To take advantage of this function, we modified DSPsim() to accept a set of coefficients as parameters rather than have them predefined, and had it calculate and return the SNR. The method for calculating SNR is discussed in section 4.3.1 SNR. We also made a new function called estimateCoeffs() which takes coefficients as its only input parameter, in one matrix, parses the coefficients, and passes them to the DSPsim() function which returns an SNR. The SNR is negated before being returned because fminsearch finds the minimum. After being returned by fminsearch, it is negated again to make it positive. The Matlab code for this function can be found in Appendix E Coefficient Optimization MATLAB Code. To automate the calculation of new coefficients for every order, we made a function called justDoIt() which uses the original coefficients obtained from the delta-sigma toolbox as "ballpark" coefficients for each order, and passes these to the fminsearch function to calculate a new set of coefficients for each order. The Matlab code for this function can be found in Appendix E Coefficient Optimization MATLAB Code. Our optimization functions were run twice. The first time, we used fewer iterations for the fminsearch function, and this took about 2 days of calculation. The second time we increased the iterations and let it run for a week to see if we could obtain better results. Indeed, we did see an improvement in average SNR, as shown in Figure 29.

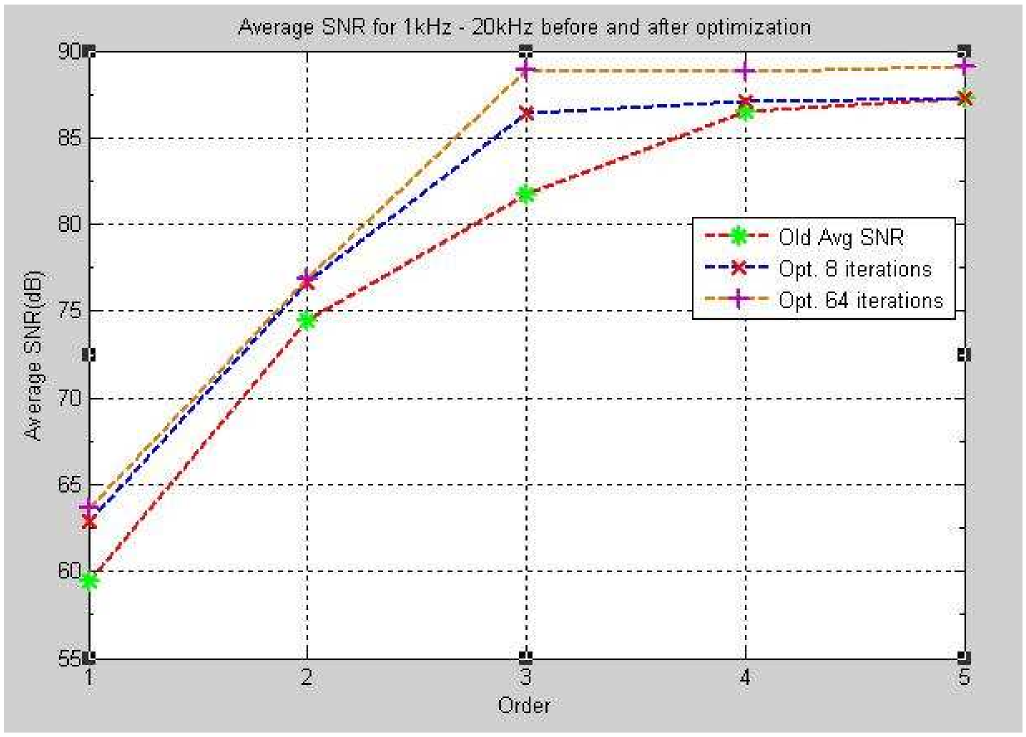


Figure 29 - Average SNR Improvement from Optimized Coefficients for Orders 1-5

Note that the above plot shows the average SNR, which is the average, for each modulator order, of all SNR's resulting from all input frequencies. Because it takes days to optimize these coefficients, we only ran this for a modulation frequency of 2MHz. It is interesting to note how average SNR saturates at around 88 dB starting from third order. This indicates that we can get away with using only a third order modulator and obtain the same SNR results, while saving processing time and having less THD than with higher order modulators. It will become apparent in the next section how there is a compromise between SNR and THD as the order of the modulator increases.

3.3.4 SNR Simulation Results

Using the new optimized coefficients, we ran DSPSim() for input frequencies ranging from 1 kHz to 20 kHz, covering most of the audio band. This was done for five different orders, and for a modulation frequency of 2MHz. Each result was passed to our THD and SNR calculation functions to yield the plots in Figure 30 and Figure 31 for SNR and THD, respectively.

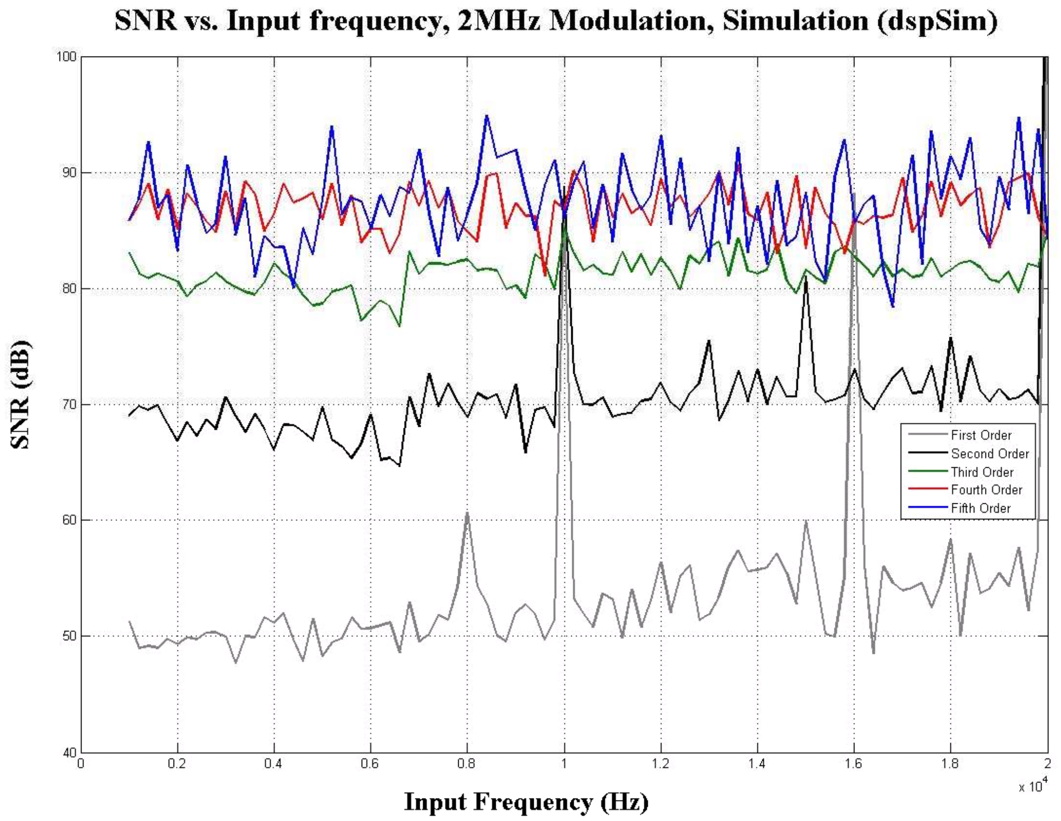


Figure 30: SNR vs. Input Frequency, Simulation

The overall trend of SNR seems to be constant as frequency increases. The average SNR increases with higher orders. Fifth and fourth order seem to be on the same level, but fifth order has greater variation, hence reaching lower lows but also higher highs, having a peak SNR at around 96dB. It is interesting to note how SNR spikes at 10 kHz for first and second order, and at 16 kHz for first order. Although it is hard to see, first and second order also spike all the way up past 90dB at 20 kHz. The reasons for these spikes are unclear.

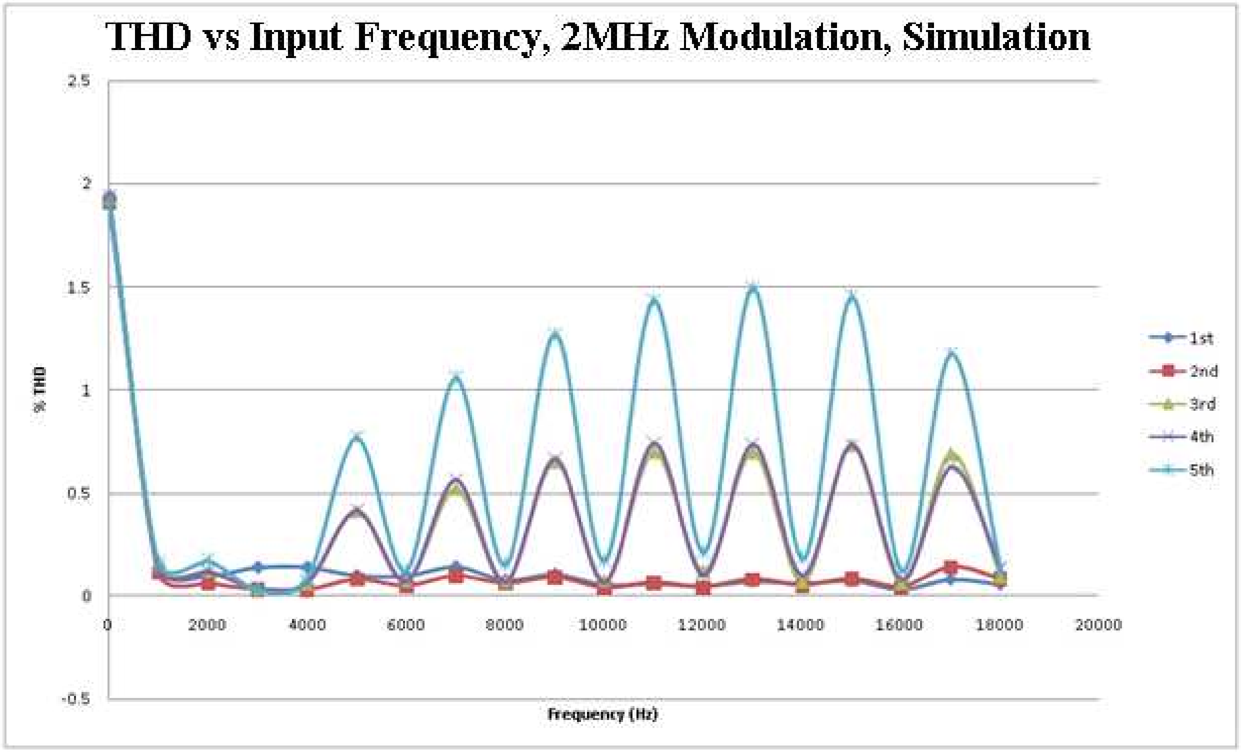


Figure 31: THD vs. Input Frequency, Simulation

Like SNR, % THD also increases as a function of modulator order. Higher THD is however undesirable, so we now see that there is a compromise between THD and SNR. The order we choose must be low enough to ensure THD is below 0.5%, and high enough to ensure SNR is above 95 dB. Let us first consider the shape of the THD plot. There is an interesting oscillation, where THD is high for odd multiples of 1 kHz, and low for even multiples. The smoothness of the lines may be misleading, because we really only have points every 1000 Hz. If we had more points, it is likely that we would see higher frequency oscillation added to the current oscillation. The overall trend of the THD also seems to be oscillating, as the peaks start low at 5 kHz, increase until they reach a peak at 13 kHz, and begin to decrease. If we had more data past 18kHz we may have seen the peaks continue to decrease, then begin increasing, following the general oscillating trend. If we were to write a formula to recreate this plot, it would be a sum of sine waves that range from low to high frequencies, where the amplitude of these waves is increased as a function of order. THD is calculated as the power of the original signal's frequency over the sum of the powers of its harmonics. Our THD plot indicates that as input frequency shifts from left to right, the original frequency power will decrease, while the sum of its harmonics' powers will increase, then the opposite will happen. What causes this behavior, however, remains unclear. The conclusion we can draw from this much speculative reasoning is that higher orders will create more radical oscillations by increasing the amplitude of all sine waves which add up to the resulting THD plot for each order in figure 35.

With respect to our desired results, if we were to take the average of all points for each order, the highest %THD will be around 0.6% for fifth order, 0.25% for third and fourth order, and 0.1% for first and second order. According to our two plots, the best choice is fourth order because it has higher SNR than third order, and only slightly higher THD than third order.

3.4 DSP Modulator Implementation

With a working DSP simulation, we were ready to implement the modulator on the DSP. However, new dimensions of complexity needed to be taken into consideration due to the nature of DSP's. This includes interrupt timing, sampling restrictions, core clock speed consideration, core hang issues, and overflow problems. A solution was found to solve each of these problems, all of which are described in this section, along with the overall system flow of our DSP program.

3.4.1 System Flow Diagram

The main difference between the DSP code and MATLAB code is that, instead of having a predefined input in a nice buffer of data, we are now sampling the input in real time. This means that we need to use timed interrupts to obtain the samples from the audio codec. The overall system diagram is simplified in Figure 32, and detailed in Figure 33.

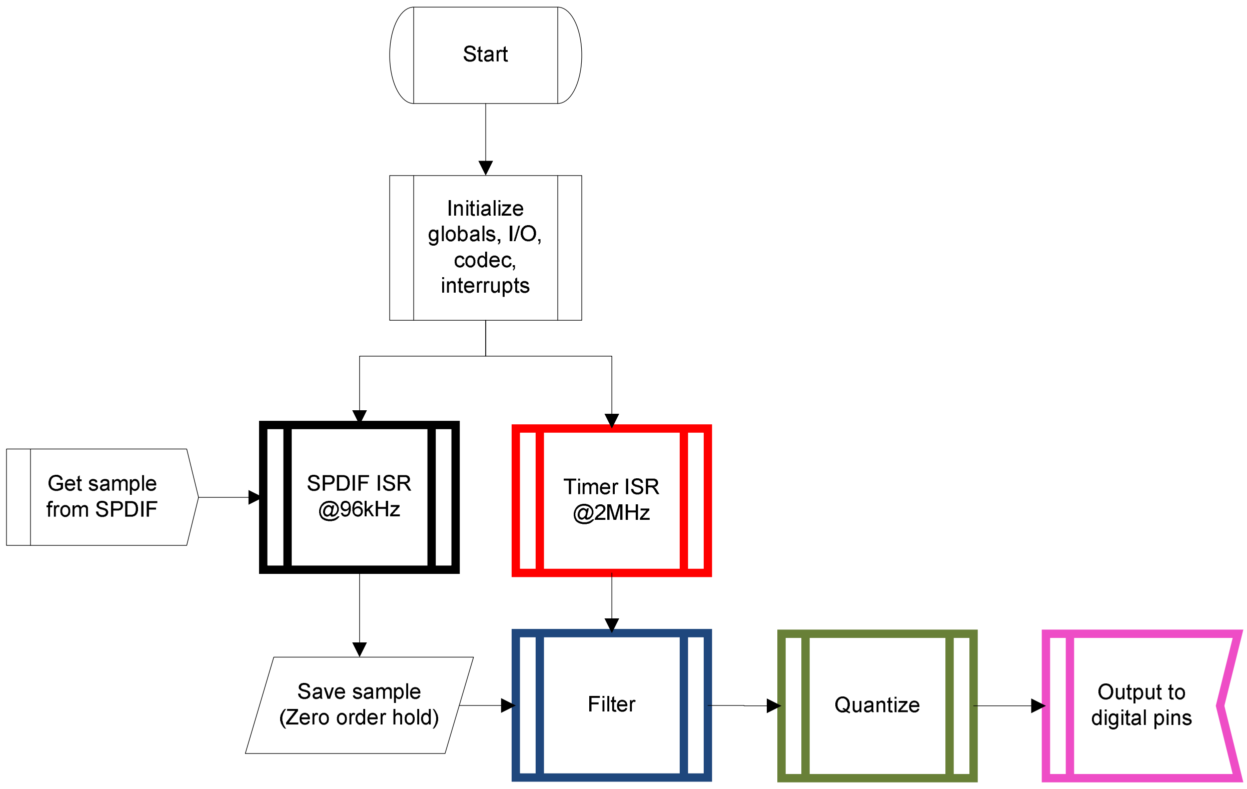


Figure 32 - Simplified System Flow Diagram

Figure 32 above is color-coded such that each colored block matches its corresponding timing diagrams shown in Figure 34 and Figure 35.

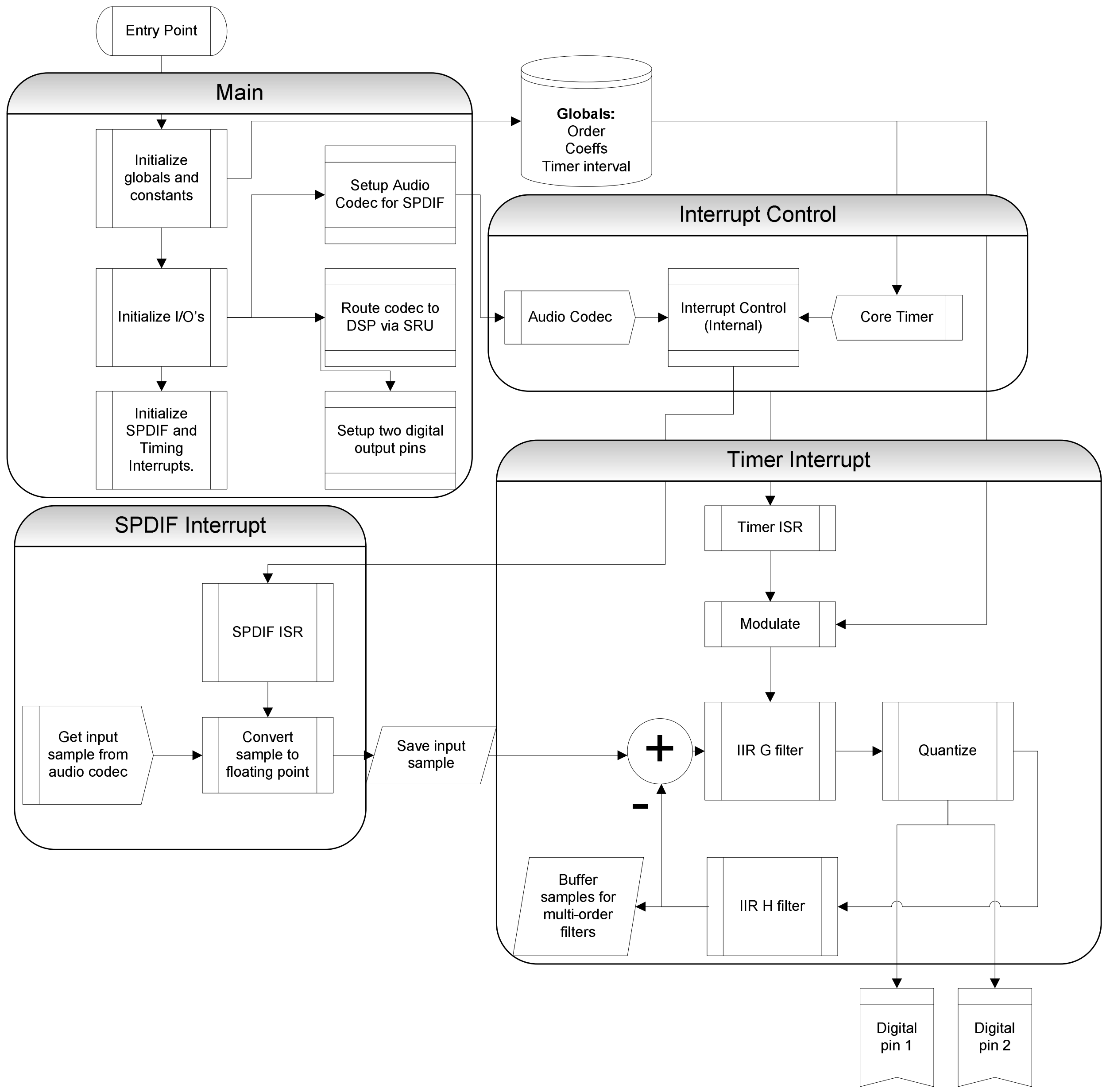

Entry Point
Main
Initialize globals and constants
Setup Audio Codec for SPDIF
Initialize I/O's
Route codec to DSP via SRU
Initialize SPDIF and Timing Interrupts.
Setup two digital output pins
Globals: Order Coeffs Timer interval
Interrupt Control
Audio Codec
Interrupt Control (Internal)
Core Timer
Timer Interrupt
Timer ISR
Modulate
SPDIF Interrupt
SPDIF ISR
Get input sample from audio codec
Convert sample to floating point
Save input sample
+
-
IIR G filter
Quantize
Buffer samples for multi-order filters
IIR H filter
Digital pin 1
Digital pin 2

Figure 33 - Detailed System Flow Diagram

The code running in each timer interrupt is almost identical to that used in our Matlab simulation. The main differences are:

- We now need to setup the hardware to read from the proper input ports and write to the desired output pins.

- The SPDIF interrupt is used to acquire data, saves this data, but does not call the timer interrupt.
- Each time the timer interrupt is triggered, it uses whatever data is available in the "save input sample" block shown in Figure 33.

The code used in the DSP is provided in Appendix G DSP Code.

3.4.2 Timing

Having a clear picture of how the timing of the input sampling relates to the timing of the core clock, and the timer interrupt, is crucial to understanding how and why the DSP program works. Figure 34 shows a sample timing scenario for a sampling frequency of 96 kHz, a core clock running at 397MHz, and a timer interrupt service routine running at the modulation frequency, or 2MHz. One can see how the modulation, which occurs once every timer ISR iteration, occurs multiple times per sample. Each time the modulation function is run (filter + quantize), the existing input sample is used, implying that we are using zero-order hold to output at a frequency of 2MHz. The reuse of input samples is acceptable in our case, because doing so will not lower our SNR below our goal.

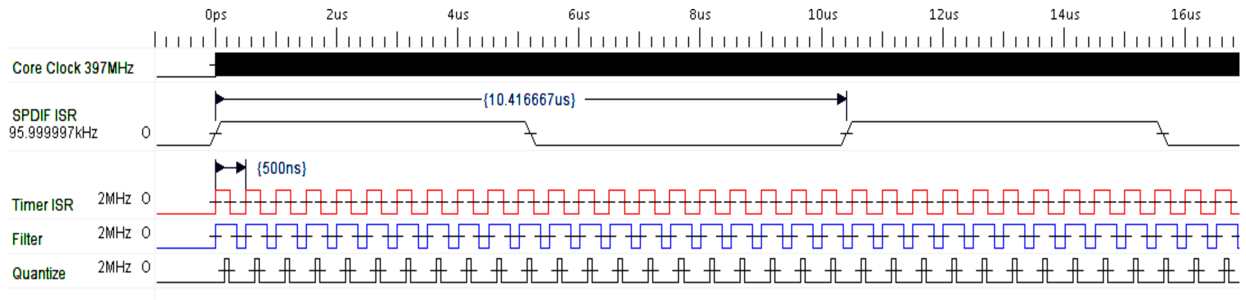


Figure 34 - System Timing Diagram

Figure 35 shows the timing dependencies and non-dependencies between the various signals and functions. What's important to note is that the modulation is independent from the sampling frequency of the SPDIF interrupt. The SPDIF ISR actually requires very few clock cycles to execute because its only function is to acquire the input sample from a register, convert it to floating point, and store it as a global variable. We have setup the two interrupts such that the SPDIF has a higher priority than the timer one. This is important because we do not want to drop any incoming samples. Once every sampling interval, the SPDIF will break the operation of the timer ISR, and update the input sample. This does not negatively affect the modulation because:

- The filter stage reads the input sample once at the beginning of its function, and does not use it again.
- The SPDIF interrupt duration is shorter than the wait time shown in Figure 35.

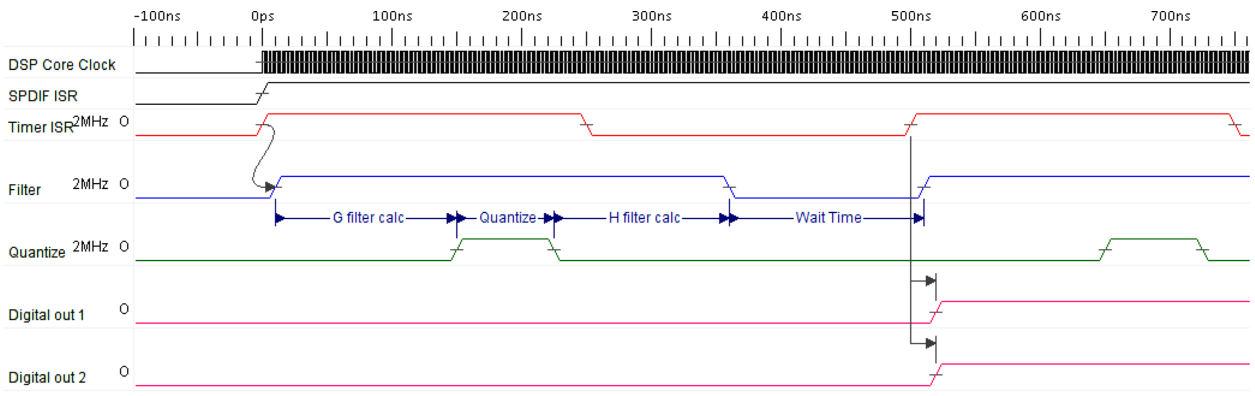


Figure 35 - Timing Relationships

As a note to avoid confusion, the timing shown for the two ISR's in the figure above does not represent the execution time of each ISR. Rather, each complete cycle represents the time between two subsequent iterations of the same ISR. This also applies to the timing in Figure 34. Furthermore, the time measures shown in the quantize line in Figure 35 are all variable. That is, the time to execute the G filter for one sample may not be the same time for the next sample. For this reason, and to account for instances where time is needed to perform the SPDIF ISR, we have a wait time that will vary from one modulation cycle to the next. The digital output pins are only updated when the next timer interrupt occurs, to make sure the outputs are equally spaced in time, and to account for randomness in modulation time.

3.4.3 Interrupts and their Pitfalls

This design calls for a significant number of operations in order to generate a single modulator output. As discussed in the previous sections, the DSP must perform multiple IIR filter operations, quantization's, and sample receiving while maintaining accurate timing between interrupts, and continued operation even at high output frequencies.

While interrupts make this type of design possible, they also add extra complications that are impossible to traditionally debug. When user interrupts are being nested, ensuring proper timing becomes difficult. The number of cycles required to initiate and return from an interrupt depends on the language and the processor being used. The number of interrupts used for this project may not be functional on other processors if the interrupt requires too many additional cycles. The particular processor used from this design had a range of interrupt types with varying cycle overhead. The fastest interrupt was not usable for the design, because some important registers were not preserved. If this design is moved to a new platform care must be taken to ensure the proper interrupt type is used in order to preserve the exact timing diagram as shown in the previous section. Errors in the interrupt order could cause dropped samples, or extended missed modulator outputs, both of which could cause audio degradation.

Not all interrupts are treated equally by the DSP. Certain system interrupts occur with a higher priority than user interrupts. With outputs in the MHz range any system interrupt, even short ones, can cause a long modulation pause. In the particular processor used for this design, the analog to digital converter interrupts had an odd behavior. The analog to digital converter uses multiple buffers. The basic operation is that the converter uses a register for the latest sample that is being received and another register for the sample that was previously received. After the previously received buffer is read, it is then rotated with the register that is used for the current sample. When the analog to digital converter generates it's interrupt the core hangs until the receive buffer is read. If the receive buffer is not read the process core will hang temporarily. This causes a pause in the modulator output, but the modulator will continue to run. This is the type of error that was common during debugging. Because the modulator continued to run, and the audio output continued to work, it was not immediately obvious that there was an output error until the output was buffer and analyzed.

3.5 Hardware and PCB Design

The design of a new PCB was vital to the implementation of full system feedback. Since this project reused the power stage and filter design from last year, much of the PCB layout was able to be reused. The PCB for this year's project was redesigned with feedback and modular connections in mind.

The initial design of the feedback network was overwhelmingly simple. The design simply added a capacitor and resistor series to each side of the output speaker, creating a DC blocked and attenuated version of the output. One half of the output signal was fed to the left channel, and the other to the right channel of the DSP's analog to digital converters. The maximum output of the amplifier was estimated as being equal to the voltage applied to the power stage. The output was attenuated to use the full scale range of the ADC for each channel. Within the programming of the DSP the two channels were subtracted, creating the attenuated output.

This setup had several problems. This method required the use of two ADC channels, and our final DSP kit only had two channels. This means we were unable to accept an analog input when feedback was implemented. This method was also lacking in its ability to reject common mode interference. The power stage of the Class D amplifier exhibits EMI that could distort the relatively low voltage feedback signal. The ADC's of the DSP are not designed to reject common mode noise, especially from such a close and powerful EMI source.

A more suitable method for feedback signal capturing utilized a differential to single ended converter built from a differencing op amp. This method still utilized a capacitor and resistor in a series combination to DC block and attenuate the single, which ensured that the output was within a range that an op amp with a +-12V supply could handle without clipping the input.

This was the method that was chosen, although a differencing op amp was not used. Instead a differential to single ended converter, THAT1200, designed to be an ADC front end with high common mode rejection and built in attention, was used.

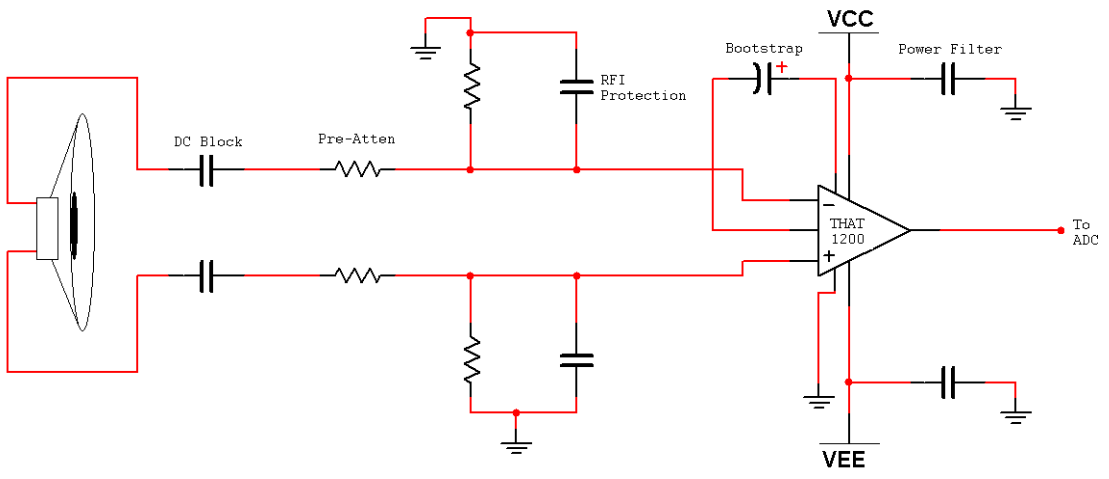


Figure 36 - Feedback Capturing Network

The PCB was designed with common mode rejection in mind. Because the DSP was used to both generate the input modulation signal and receive the feedback signal, both these I/O's were placed on the same side of the board. This meant that the feedback signal would have to travel the entire length of the board and cross a high power copper plane split. The signal was kept differential until the very end of the board, ensuring that the ADC driver could reject the noise added by the power stage.

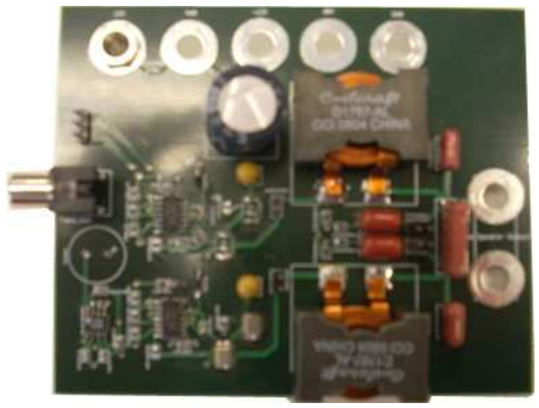


Figure 37 - Class D Audio 2009 PCB

4 Testing

To ensure that the design met its goals, a series of tests were developed to acquire a wealth of data points from the system. Testing was performed using a mix of hardware, such as scopes and precision power supplies; and software, like MATLAB. Hardware testing was straightforward, and did not involve any elaborate setups. However, software testing, in MATLAB, required complicated code.

There were three main categories for testing. The first category was power and efficiency. This was easy to test as it only required the use of some basic lab equipment, careful observation of the measurements, and some simple math. The second category was input testing. This was even easier to test as it only required finding appropriate test sources to provide digital or analog inputs. The third was sound quality. Testing sound quality required audio capturing as well as software based computation of the signals.

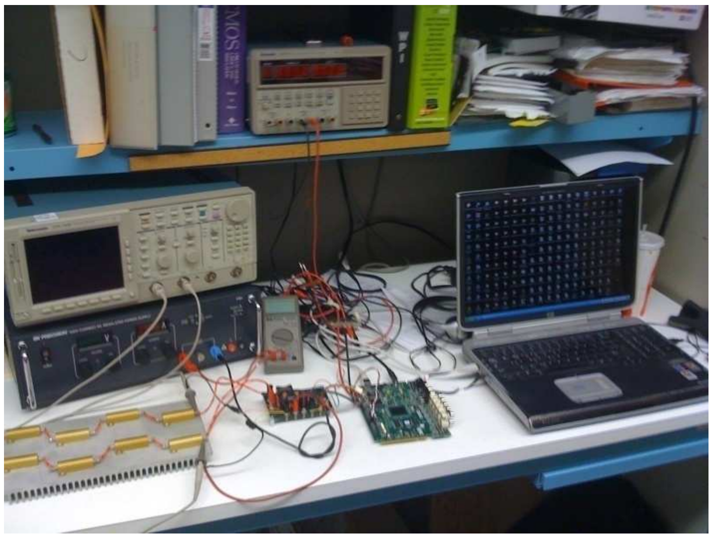


Figure 38 - Test Bench (Left to Right: Resistor Test board, Amplifier PCB, DSP)

4.1 Power and Efficiency

The setup for testing the power output of the system was simple. The output power is calculated using the output amplitude, measured at the speaker, as well as the resistance of the speaker itself. Because testing lasted a long time, a series of resistors, totaling 8-ohms, was used as the output load as opposed to a speaker. Placing a speaker on the output of this 80+ Watt amplifier would have been very disruptive when running at full power. The exact resistance of the speaker was measured using a precision Hewlett Packard 3458A multimeter. The output, RMS voltage, was measured using a Tektronix TDS784C oscilloscope.

Testing for efficiency was slightly more involved. The efficiency of our system was determined by the ratio of its input and output power. The PCB for the system called for two power rails. The first was a low voltage, low current rail required to power the gate drivers. The gate drivers were isolated from the actual amplification process because they only made contact with the near infinite input resistance of the MOSFET gates; as a result their input power did not fluctuate greatly.

The second rail powered the amplifier's MOSFETs. At any given time, current delivered to this rail will have passed through two MOSFETs, two inductors, and the load (speaker). The power delivered by this rail was the overwhelming contributor to the input power calculation. The voltage applied to the rail directly affected what output power is achievable.

The rail for the gate driver was powered by a common Tektronix PS2521G programmable power supply, as its requirements were not out of the ordinary. This supply offered a two digit current display which was used to calculate the input power for the gate drivers. The rail for the MOSFETs required a high power supply so the BK Precision High Current DC Regulated power supply was used. This power supply only had a single digit current display, so an in-line Hewlett Packard E2373A multimeter was used to measure the supplied current.

4.2 Input Sources

One of the goals for this design was to be able to support both analog and digital inputs, so both of these needed to be tested. The analog input for the system was tested using a function generator. The input was tested for a series of frequencies and amplitudes, ranging the width of the audio band and the peak input of the ADC. Subjective listening tests were also performed with devices like PC's and iPods. The digital input was tested using a DVD player with an SPDIF output. A CD was generated that contained a similar range of frequencies as the analog test, but it was limited to a single amplitude. Subjective listening tests were also performed using the same DVD player and music CD's.

4.3 Audio Quality

Testing audio quality was much more challenging than the previous tests. Audio quality measurements include SNR, zero-input SNR, and THD. Measuring these involved recording the output of the amplifier for various inputs and determining the SNRs and THD of these measurements.

The audio was captured using a 192 kHz 24-bit sound card, the EMU-0202. This sound card had a limited input range of approximately 2.7 Vpp, so the output needed to be attenuated before recording, as the input of the sound card would be constantly peaked otherwise. The attenuation was performed by the feedback driver built onto the PCB. While feedback was never implemented in the DSP, the hardware portion was built, and this served to both attenuate the output as well as convert it from a differential to a single ended signal. The audio signals were recorded into SoundForge6, a piece of software capable of recording with a sampling rate of 192 kHz.

The test input signals for the system were generated within the DSP. The primary input of the DSP is a digital source, which experiences no distortion due to noise, so there is no difference between inputting a sine wave from a digital source, and generating the sine wave within the DSP during the digital signal buffer's receive interrupt. This allowed greater control of the test frequency pattern.

The testing process was automated within the DSP. After loading a specific set of modulator coefficients, and settings the desired modulation frequency for the modulator, the DSP would begin to output 1 second clips of 1 kHz spaced frequencies. After ranging the input from 20Hz to 20 kHz, the order of the modulator was increased. This allowed the testing of nearly 100 different input frequency and modulator order combinations in just under a minute. Recordings were captured for three different modulation frequencies, 5 different orders per modulation frequency, and 15 frequencies per order. After the recordings were taken they were sent to MATLAB for processing.

4.3.1 SNR

In MATLAB, several functions were written to find the zero input SNR, SNR, and THD of an audio clip and to automate the process of calculating these statistics for all of the different data that we had collected. First, the method used for calculating both SNRs will be examined. As was discussed in section 2.2.3, SNR is defined as in Equation 6 below.

$$SNR = 10 \cdot log10\left(\frac{sigPWR}{noisePWR}\right)$$

Equation 6 - Definition of SNR

There are two different ways that the power of the noise can be determined. For zero input SNR, the power of the noise is found by recording the output of the system when the input is zero. For SNR, the noise is anything in the recorded output that is not the desired signal. We chose to calculate both of these parameters in order to compare them to each other and the results of other Class D MQP projects. First, our method of calculating zero input SNR will be examined.

The noise of the system for the zero input SNR was taken as a recording of the amplifier's output while the input to the amplifier was a mathematical zero. The noise power was calculated directly from this recording. To find the signal power for this case, separate recordings for various different input frequencies were acquired. The power of the desired signal in these recordings was taken as its maximum dB magnitude value in the frequency domain. After calculating these two parameters, the formula from Equation 6 above was used to determine the SNR. The MATLAB code used to calculate the zero input SNR can be seen in appendix F. Although the zero input SNR gives us a good metric for comparison to previous MQP projects, the process of determining a more realistic SNR will be examined next.

In order to calculate the SNR, both the power of the signal and the power of the noise must be calculated again. The problem is that both the noise and the desired signal are combined in the output. Luckily, we know that the desired signal, or what the output would look like if the system added no noise at all, is a pure sine wave. This fact allows us to separate both the noise and the desired signal from the recorded output. Because the amplifier is a linear system, the desired output sine wave must only be a shifted and amplified version of the pure input sine wave. The general idea is to create the desired signal from the output, use that to find the signal power, and subtract it from the recorded output to find the noise and hence the noise power.

In order to create the desired signal sine wave, we had to create an amplified, time delayed version of the input signal. Shown below is an equation for this sine wave, with Â as the amplification factor, ố as the delay factor, y as the desired output, and cos(t) as the input sinusoid.

$$y = \hat{A} * \cos\,(t + \acute{\hat{o}})$$

Equation 7 - Desired Sine Wave Output

The values of Â and ố for any given signal can be estimated as shown in Equation 8 and Equation 9 below, respectively (Kay, 1993). In these equations, y' is the given signal, f is the frequency of the signal, n is the number of samples in the wave, and Fs is the sampling frequency of the wave.

$$\hat{A} = \frac{\left| y' * e^{\frac{-j*2*\pi*f*n}{Fs}} \right|}{n * 2}$$

Equation 8 - Formula for Estimating Amplification Factor Â

$$\acute{\hat{o}} = \frac{\tan^{-1}[-y' * \sin\left(\frac{2*\pi*f*n}{Fs}\right)]}{y' * \cos\,(\frac{2*\pi*f*n}{Fs})}$$

Equation 9 - Formula for Estimating Time Delay Factor ố

After finding the values of Â and ố, the desired output can be built according to Equation 7 above. Calculating the power of this desired output yields the signal power. Subtracting the built signal from the actual output yields the noise signal. The noise power is calculated from that signal. Once the power of the desired signal and the noise is known, these values are plugged into the formula for SNR from Equation 6. The actual MATLAB code used to implement this method can be found in appendix F.

4.3.2 THD

Total harmonic distortion is defined as the ratio of the power of a signal's fundamental frequency to the power of its harmonics, expressed as a percentage. To obtain results that are comparable to similar Class D projects from previous years, we will only measure the first five harmonics. If we let P_f be the power of the fundamental frequency, and P_n be the power of the n^th^ harmonic of that signal, then THD is calculated as follows:

$$\%THD = \frac{P_f}{\sqrt{{P_1}^2 + {P_2}^2 + {P_3}^2 + {P_4}^2 + {P_5}^2}} * 100$$

Equation 10: THD formula

To obtain the power of the fundamental frequency and its harmonics, the first step is to take the FFT of that signal. The power of any frequency within that FFT is simply its magnitude. In Matlab, we obtain the power of the fundamental frequency by finding the max value in the FFT. The index of this value is then multiplied by *n* to give us the index of the *n*th harmonic, which we then use to retrieve its value from the FFT array. We do this for the first five harmonics, square them, and take the square root of their sum, by which we divide the power of the fundamental frequency, and multiply the dividend by 100 to yield the final % THD. For the Matlab code used to calculate THD, refer to appendix F (getStats.m).

4.3.3 Automating Sound Quality Testing

To obtain an adequate amount of data for analysis, while minimizing manual labor to obtain it, we automated both the generation and acquisition of a range of output data from the DSP, the measurement of the results, the formatting of the results, and the plotting. First let us consider the variables we are dealing with. We want to test SNR, THD, and efficiency for three different modulation frequencies: 1MHz, 1.5MHz, and 2MHz. We chose these values because our DSP cannot modulate faster than 2MHz due to timing restrictions, and 1MHz is already below 60x oversampling requirement. For each of these, we need to test five different modulator orders. Fifth order is the maximum our DSP can support. A higher order would require more time than allowed by the slowest timer ISR at 1MHz. For each order, we want to test enough frequencies to generate a meaningful plot. We chose to test for nineteen different frequencies, equally spaced along most of the audio band, from 20Hz to 19.20 kHz. We now have three dimensions: modulation frequency, order, and frequency. The total number of SNR calculations needed is 3*5*19 = 285. The same goes for THD.

Automating the Generation of Test Signals:

Instead of actually sampling an SPDIF input for 285 different possibilities, we decided to generate each within the DSP. This does not compromise the credibility of the results because an SPDIF signal is a digital signal not unlike the digital signal we generate in the DSP. To stay true to the interrupt restrictions, we simulate sampling by incrementing a time variable by 1/96 kHz every time the SPDIF interrupt is called, and take the sine of this value times the desired frequency to produce a sample, as seen from the following DSP code extracted from the sampling interrupt:

```
timeADC+=time_diff;     //time_diff = 1/96000
freqNum = timeADC/0.5;  //increment frequency index every 0.5 seconds
adc = cos(2*PI*freq[freqNum]*timeADC); //generate sample
```

In the main function, we create an array freq[] than contains 20 different frequencies. These frequencies start at 20Hz, and increase by 1000 every iteration until they reach 18.20 kHz, for a total of 19 different frequencies. In the sampling ISR, we fetch a new frequency from freq[] every 0.5 seconds. In the timer ISR, we check if freqNum, the frequency counter, has passed 19, in which case we reset it to 0 and increment the modulator order. As a result, if we let our program run, it will modulate and output 19 frequencies per order, with a duration of 0.5 seconds per frequency, for 5 different orders. Our manual task is to change modulation frequency twice. With this, we were able to easily let the DSP play and record the filtered output, then split up the recording into multiple wave files for each frequency.

Automating the Measurement of the Test Signals:

Having generated and recorded all the necessary test signals, we now have to measure the SNR and THD of 285 different wave files. This process had to be automated as well. The first step was naming the wave files such that we could easily parse the file names to determine its order, modulation frequency, and signal frequency. We then wrote a function called getStats(), which uses the following logic to calculate the THD and SNR of each of these wave files:

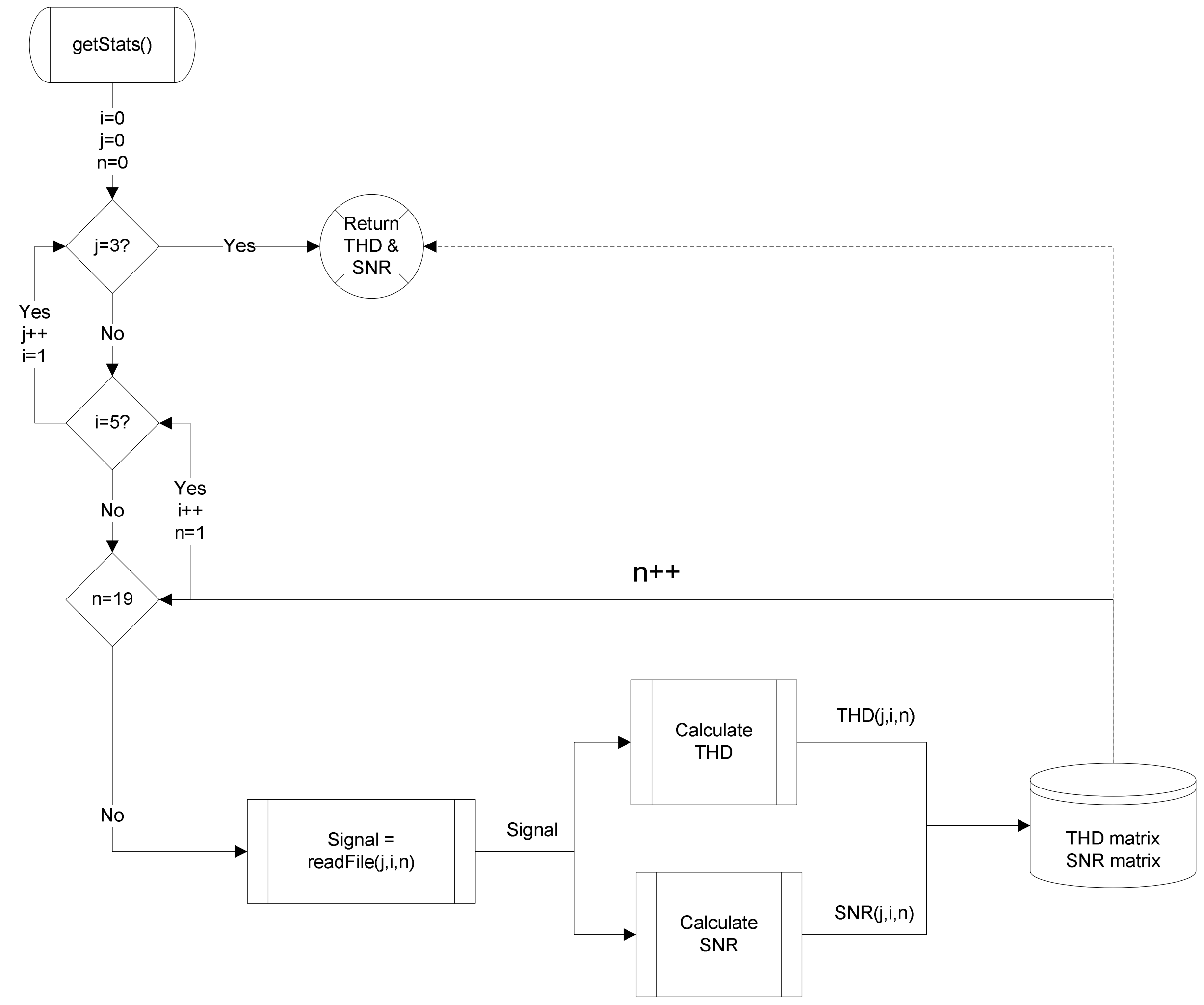


Figure 39: Flow Chart for Automating SNR and THD Calculations

The overall idea is to have three levels of loops, j for the three different modulation frequencies, i for the five different orders, and n for the nineteen different frequencies. Within each iteration, the function readFile() will be given the current value for i, j, and n, which will let it determine which file to read and return. The readFile() function will use the given i, j, and n to generate the name of the file according to the naming convention we used, then save it in an array called signal and return it. This array is then passed to a function to calculate SNR, and then a function to calculate THD. When SNR is calculated, the result is saved in a three dimensional array, with the indices being the current i, j, and n. The same goes for THD. When all loops have terminated, the function returns these two arrays. The code for getStats() and all the functions it calls is found in Appendix F.

We must note that a slightly different version of the getStats() function, called getZeroSNR(), was used to measure the SNR of zero-input measurements, using the zero-input SNR calculation method. This introduced a fourth dimension to our array of results, because in addition to 3 modulation frequencies, 5 orders, and 19 frequencies, we now had 3 different types of zero input measurements. The first type was the measurement of noise with the modulator powered on but with no input given to it. The second type was the noise measured with the modulator powered off. Finally, we measured the noise with only the feedback attenuation circuit powered on. The difference between getStats() and getZeroSNR() was the SNR calculation function used, an additional file being returned from the readFile() function, called noise, and a fourth loop to encompass the 3 different types of zero input. The results of getZeroSNR() did not include THD, and was a four dimensional array of SNR measurements. Its code is also found in Appendix F.

Automating the Parsing and Plotting of Measured THD's and SNR's

We now had three-dimensional matrices for both THD and SNR, and a four-dimensional matrix for zero-input SNR. This made the task of plotting the results rather complicated. We therefore wrote a function to reformat these matrices into structs that included 15 by 19 arrays for THD, SNR, and frequency. The 19 columns represented the 19 different frequencies, the first 5 rows were orders 1 through 5 for a modulation frequency of 1 MHz, the next 5 rows for 1.5 MHz, and the final 5 rows for 2 MHz. The function used to format was called format() and is found in Appendix F under format.m. Finally, we wrote a function called plotForUs(), which took any two formatted results, and plotted one against the other on a single plot, labeling the axes and title accordingly, adding a legend, and color coding the results for the five different orders. This allowed us to plot, using a single function call, frequency vs. SNR, frequency vs. THD, or even SNR vs. THD, although the latter did not prove to be of any use. The code for this function is found under plotForUs.m in Appendix F.

5 Results

The goal of this project was to build an 80 Watt amplifier with both a digital and analog input that could achieve over 95% power efficiency, more than 100 dB SNR, and less than 0.5% total harmonic distortion. Our final amplifier design met every aspect of this goal, making the project a success. In this section, the detailed specifications of the amplifier will be examined.

The specifications do not explain how we met one of our goals, to build an amplifier that had both digital and analog inputs. However, this goal was met and tested thoroughly. After the amplifier design was finalized, a DVD played with an SPDIF (Digital) output was used as the input to our amplifier. The final test for this functionality was run by playing an audio CD in that DVD player and listening to the output of the amplifier by attaching a speaker to the load. The music that played was qualitatively "good" sounding. The final test of the analog input was done using the same DVD player's analog output, which produced an output with the same level of quality as the digital input. The data used to calculate the SNR, THD and efficiency power results that follow was collected while a digital signal from inside the DSP was used as the input to the modulator.

5.1 SNR

As was discussed in the testing section of the report, there are actually two methods of calculating the signal to noise ratio of the amplifier for any given input. Initially, our SNR goal of 100 dB was written with the intention of improving upon previous Class D audio amplifier MQPs. Because the official SNR used for last year's Class D project was zero input SNR, we calculated the same parameter for a comparison. The results of this test were even better than achieved in previous years. As can be seen in Figure 40, Figure 41, and Figure 42 below, the average zero input SNR of our modulator was roughly 109 dB for all orders and modulation frequencies. The zero input SNR does drop significantly at an input frequency of 18 kHz for every modulation frequency. The raw data used to create the zero input SNR figures below can be found in appendix H.

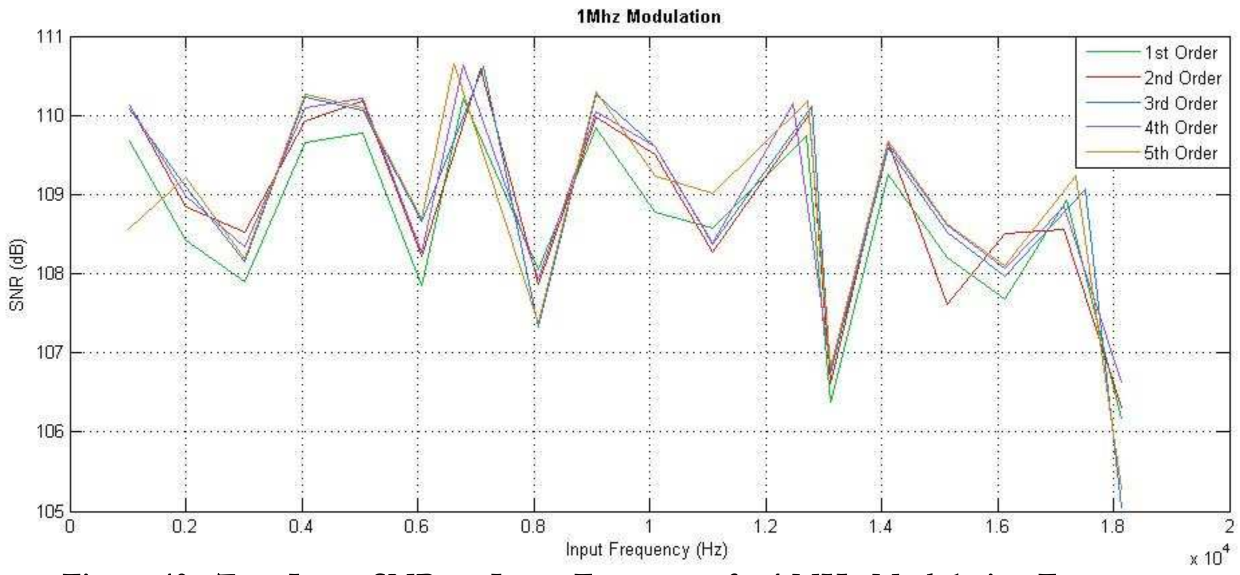


Figure 40 - Zero Input SNR vs. Input Frequency for 1 MHz Modulation Frequency

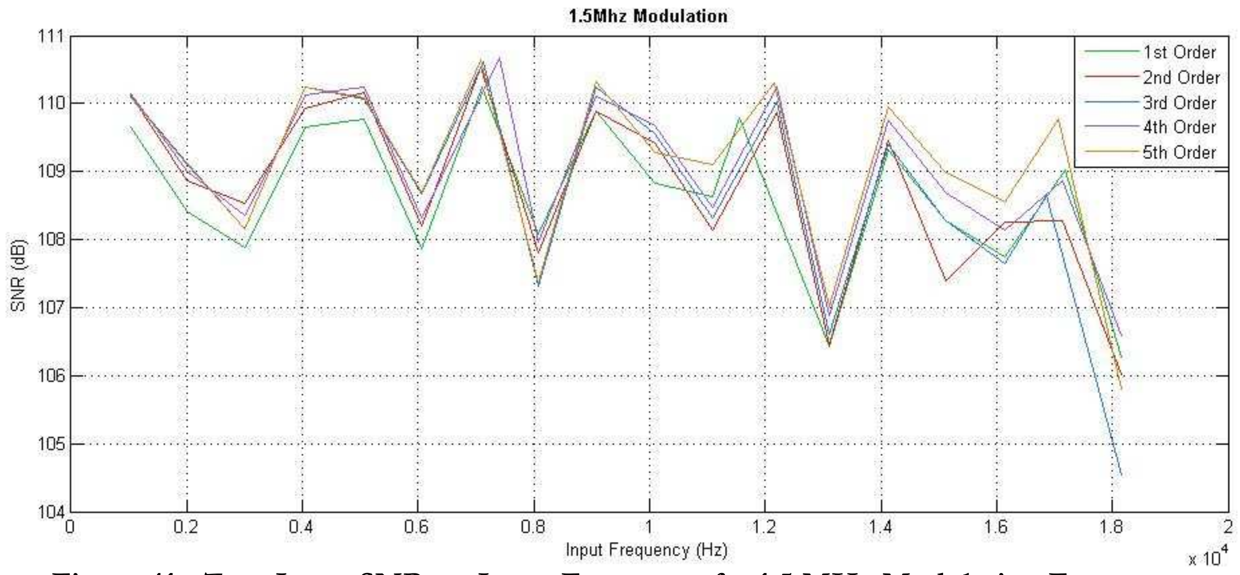


Figure 41 - Zero Input SNR vs. Input Frequency for 1.5 MHz Modulation Frequency

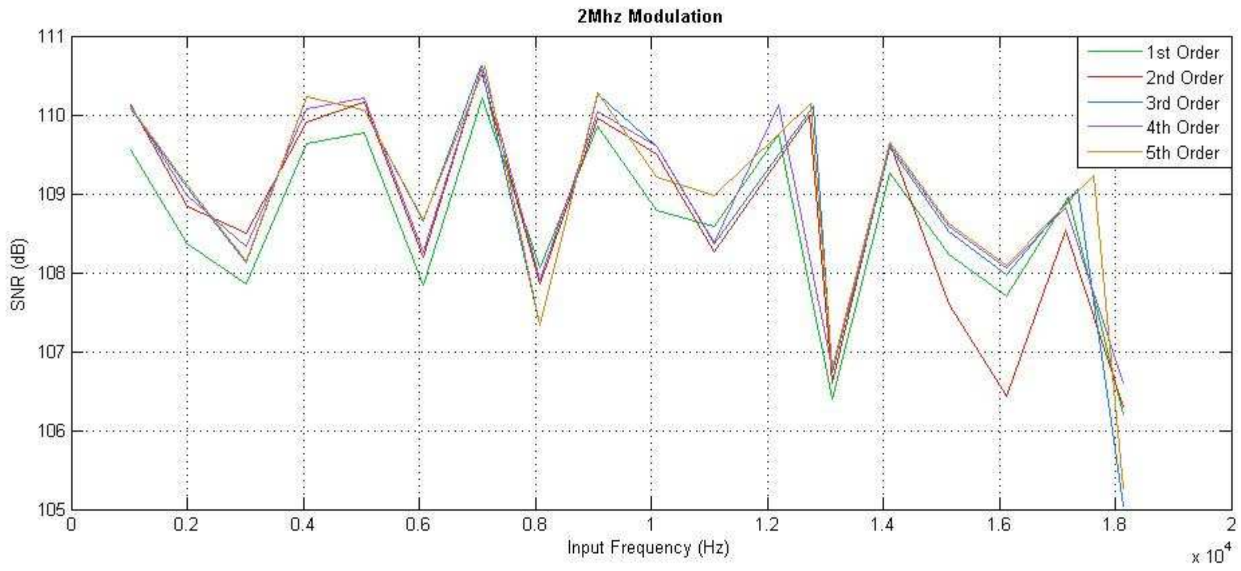


Figure 42 - Zero Input SNR vs. Input Frequency for 2 MHz Modulation Frequency

These zero input SNR numbers easily accomplish our goals, but do not make much sense in terms of the qualitative definition of signal to noise ratio. Because our amplifier is driven by a digital signals processor, the amplifier can effectively be turned off every time the input is zero. This fact made our zero input SNR values somewhat non-descriptive of the actual performance of our system. In order to more effectively measure the noise reduction of our amplifier, the actual SNR of several different output samples were calculated. This data is displayed below in Figure 43, Figure 44, and Figure 45. There are two main trends in these figures. The first and more obvious one is that SNR is steadily decreased with increasing frequency. This makes sense, because as the input frequency goes up, the oversampling ratio of the delta-sigma modulator goes down and is therefore less effective at approximating the input. This steady decline is impressive, because each of the plots contains data from 18 different input frequencies. The second trend is the fact that higher order modulators seem to produce worse SNR results, with the exception of first order. The fifth order modulator produced the overall worst SNR results, which is counter intuitive. The first order modulator also performed very poorly, which was expected. The raw data used to create the figures below can be found in Appendix I.

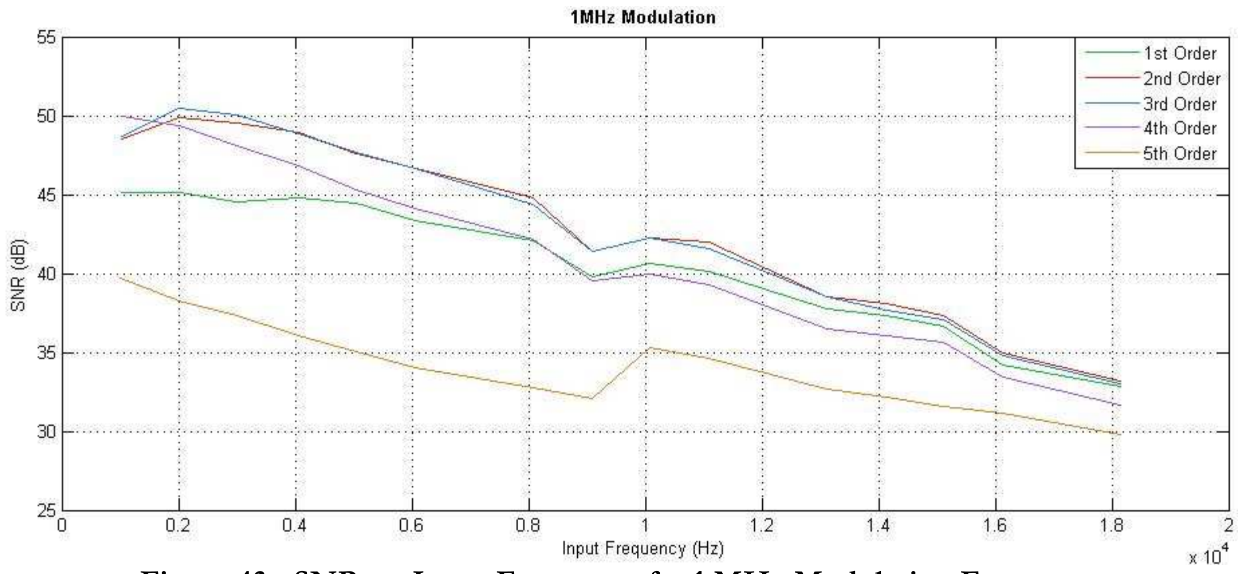


Figure 43 - SNR vs. Input Frequency for 1 MHz Modulation Frequency

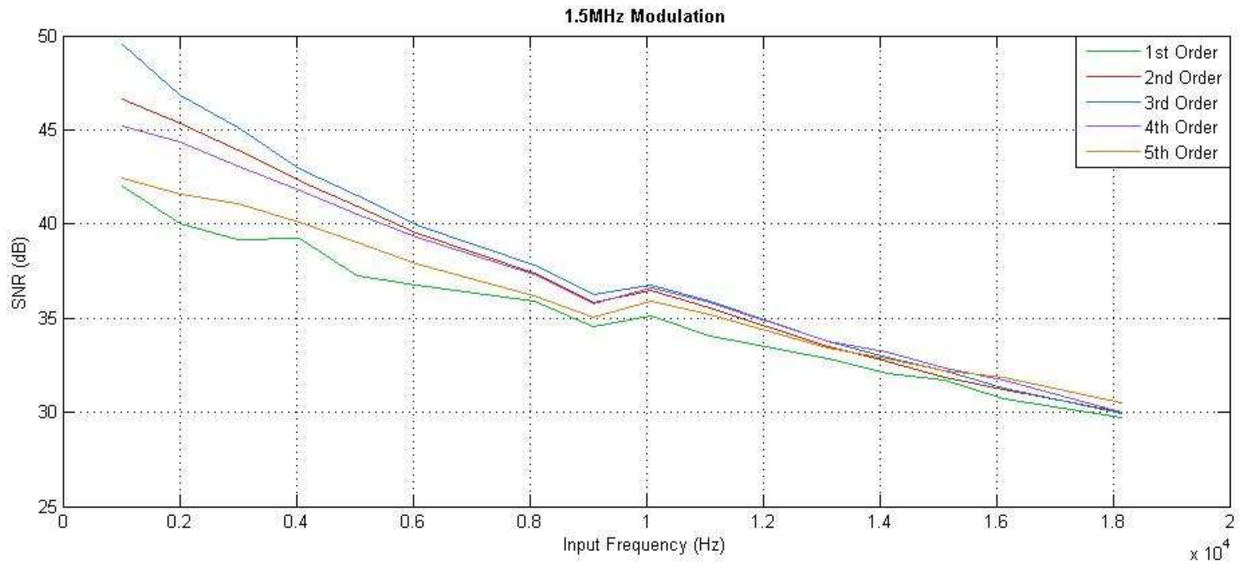


Figure 44 - SNR vs. Input Frequency for 1.5 MHz Modulation Frequency

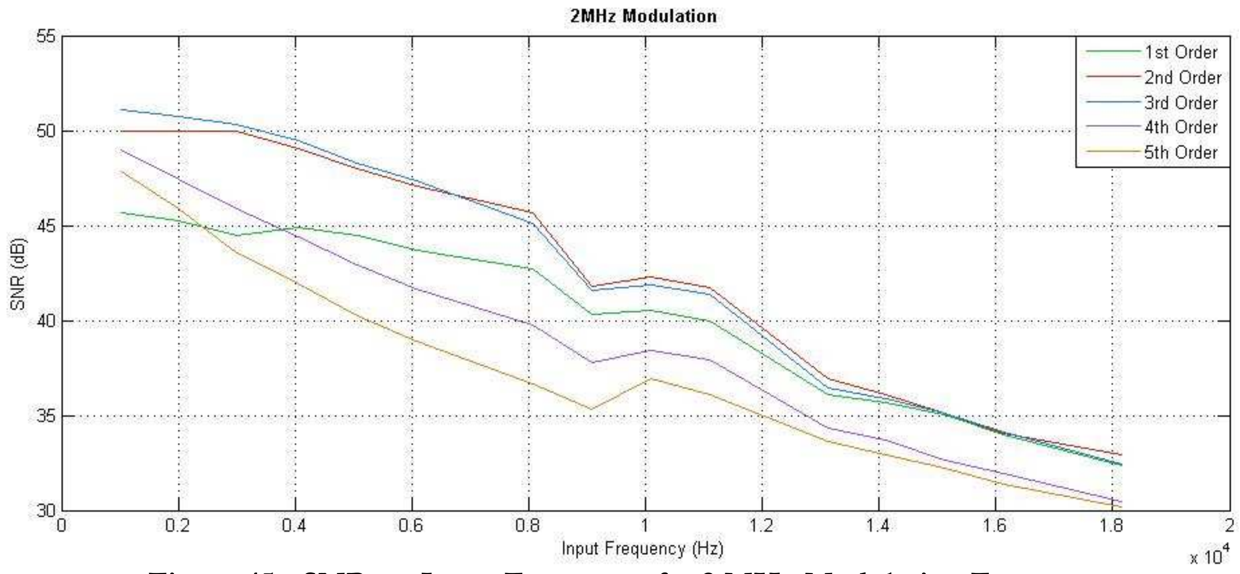


Figure 45 - SNR vs. Input Frequency for 2 MHz Modulation Frequency

5.2 THD

The results of the THD testing also yielded numbers below our goal of 0.5%. As can be seen below in Figure 46, Figure 47, and Figure 48 below, the THD numbers for all five orders are almost always under 0.5%. There are exceptions, such as certain input frequencies when the amplifier was running at 1 and 1.5 MHz modulation frequencies and using first order coefficients. At a modulation frequency of 2 MHz, however, this pattern seems to break and it is the fifth order modulator which yields the worst THD numbers, followed by forth order. The fifth order modulator even produces THDs above 1% for a few input frequencies, which is much higher than seen under any other circumstances. The raw data collected to create the figures shown below can be found in Appendix J.

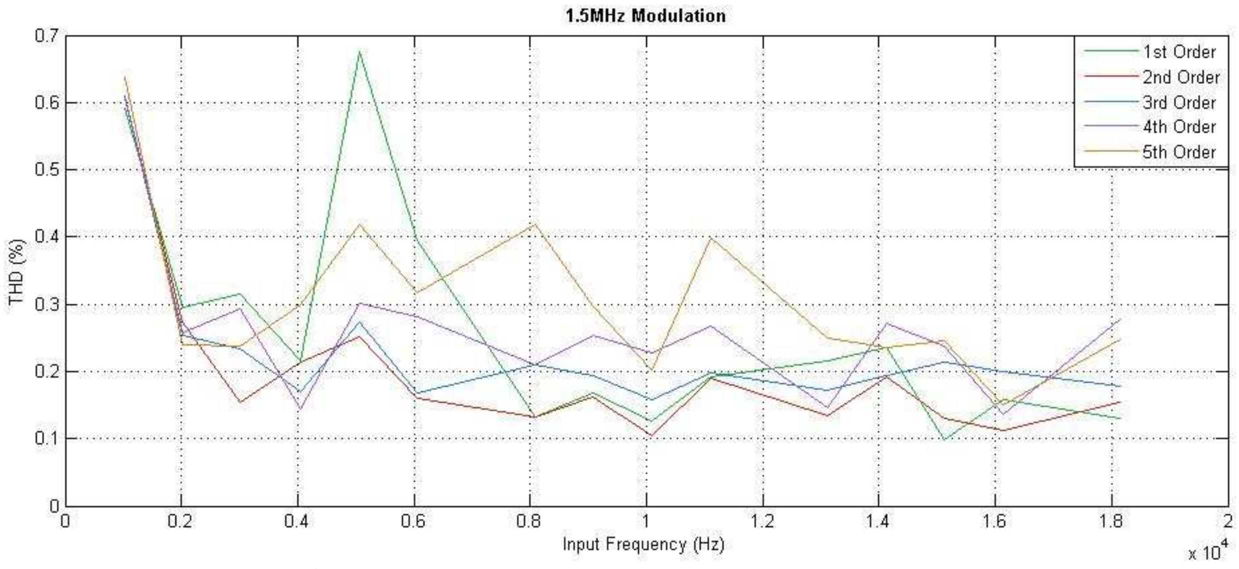


Figure 46 - THD vs. Input Frequency for 1 MHz Modulation Frequency

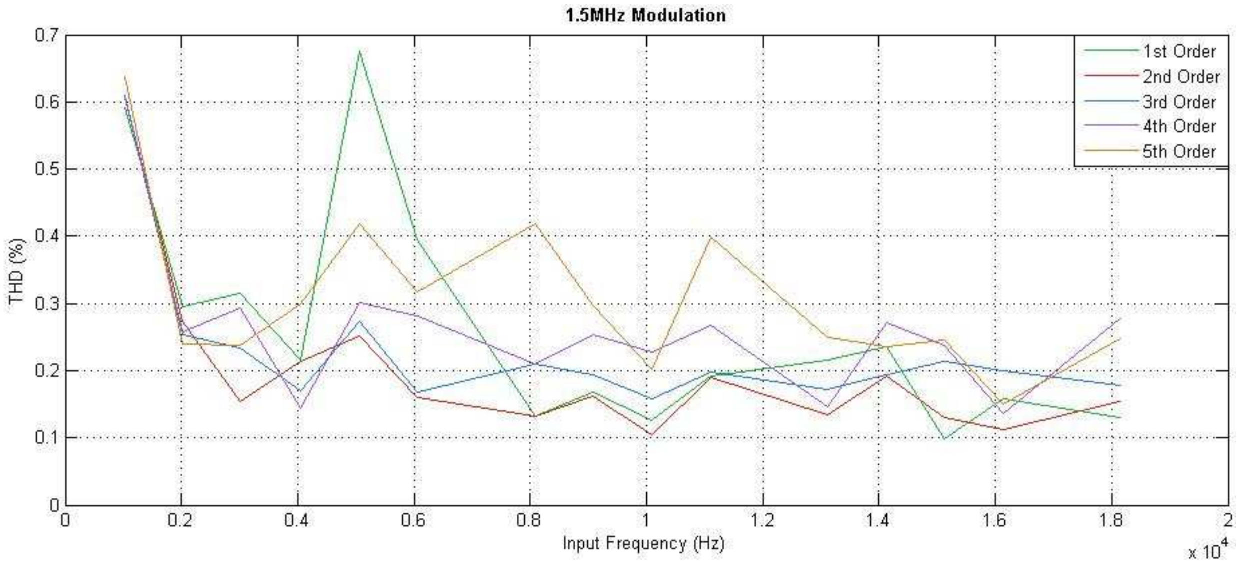


Figure 47 - THD vs. Input Frequency for 1.5 MHz Modulation Frequency

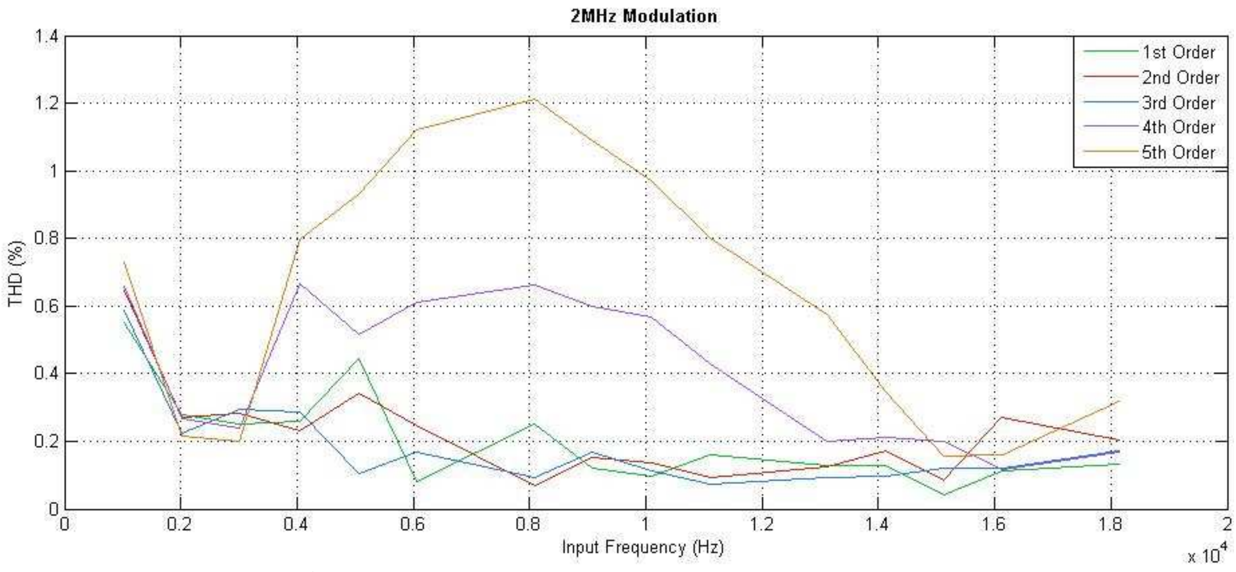


Figure 48 - THD vs. Input Frequency for 2 MHz Modulation Frequency

5.3 Efficiency and Power

Efficiency and power are mostly dependant on the power stage and filter, but there is some influence from the modulator scheme. In order to determine the influence that a modulator has on the efficiency and power output of a Class D amplifier, test points were accumulated for multiple frequencies, with multiple modulator orders and modulation frequencies. The raw power and efficiency data that was collected can be seen in appendix K.

The power output goal of the project was clearly met, reaching values above 90 watts. The figure below shows the output power versus the input frequency for two different modulator orders. In both cases the modulator frequency was locked at 2 MHz, and the input was locked at an amplitude of 2. Since the input is digital, the input amplitude is rather vague and the importance of the input amplitude is the relative size. And input amplitude of 2 is the largest input that was used, larger values become unstable for many modulator orders.

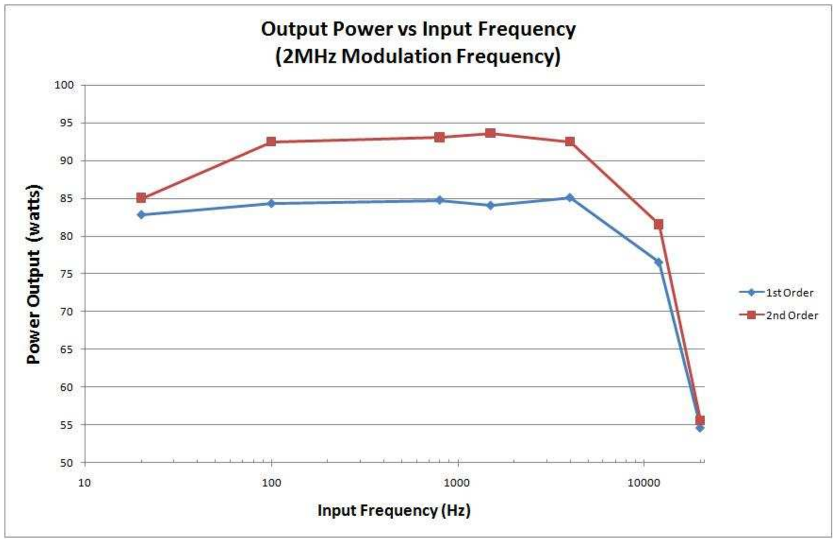


Figure 49 - Output Power vs. Frequency (2MHz modulation frequency, high input amplitude)

Some of the higher order modulators became unstable at higher amplitudes. In order to get a relative scale of power output versus modulator order, another data set was created using a lower input amplitude. This resulted in lower output power, but gave better insight into the differences between the orders. From the following plot two possibilities result. The first is that the test data for the 1st order modulator was somehow skewed, resulting in a very low reading. The other possibility is that there is an exponential relationship between modulator order and output power which is asymptotic to a certain value.

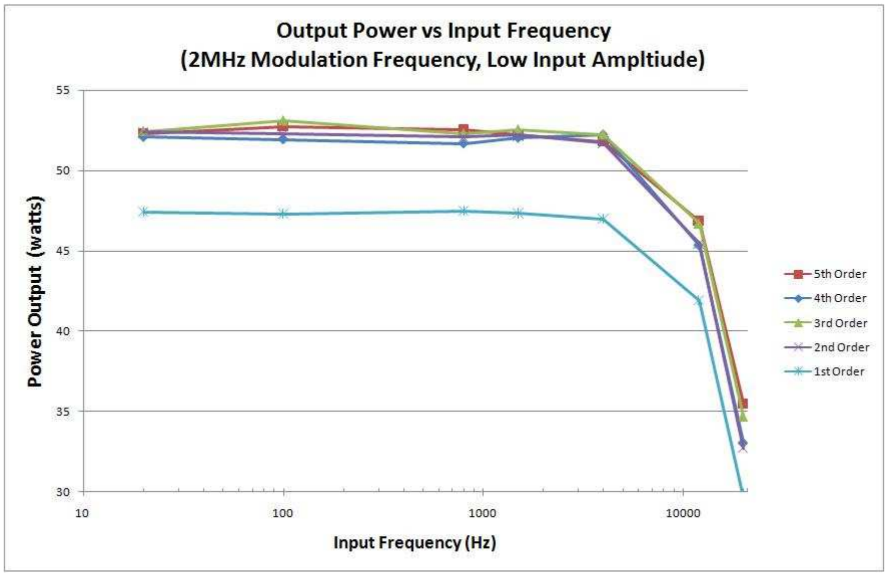


Figure 50 - Output Power vs. Frequency
(2MHz modulation frequency, medium input amplitude)

The results for efficiency proved to be equally interesting. The figure below shows the average efficiency for each modulator, with an input amplitude of 1.5 and a modulation frequency of 2 MHz. The first three orders show the efficiency of the system asymptotically approaching 95.5% by the third order. However; the 4th order efficiency results take a sharp dive to below 94% efficiency.

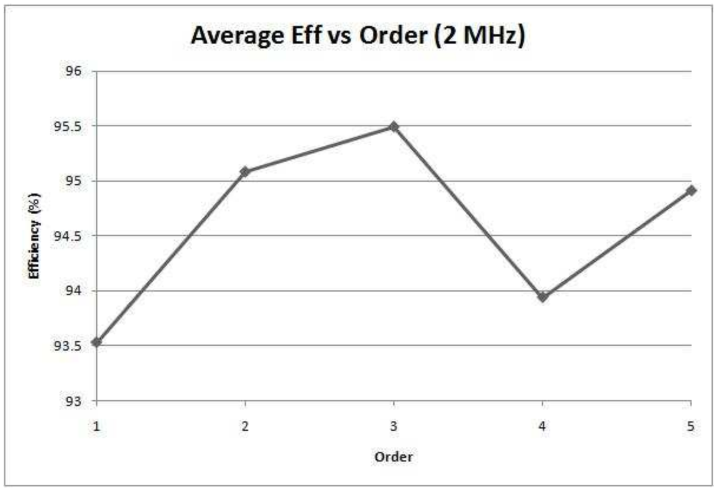


Figure 51 - Average Efficiency vs. Modulator Order
(Input amplitude 1.5, 2MHz modulation frequency)

The degree to which the 4th order efficiency drops appears to be a result of some outlier data points, namely 20 kHz. For some orders the efficiency at 20 kHz was very similar to the efficiency of the other frequencies. In the figure below the, efficiency for only the 20 kHz data point for each order is shown.

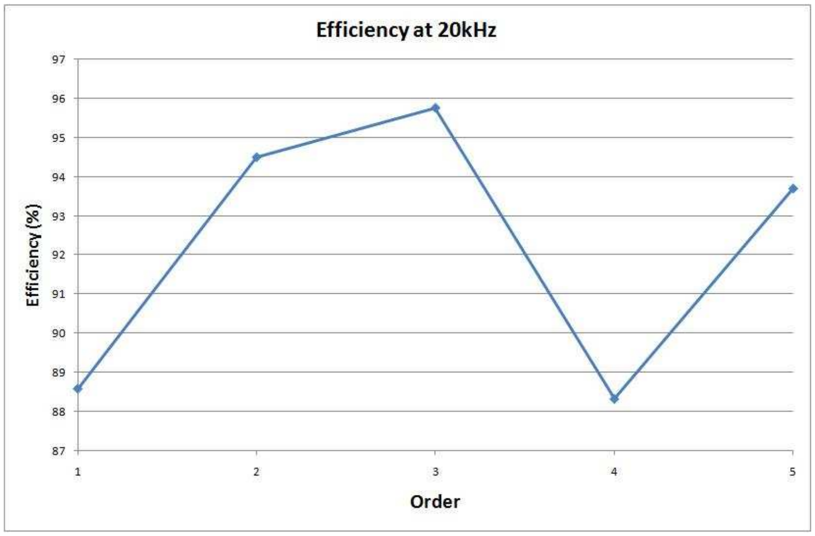


Figure 52 - Efficiency for All Modulator Orders at 20 kHz

It is clear from this figure that the average efficiency is dominated by this point. There also appears to be some correlation between the 1st and 4th orders with regard to the efficiency of the system. The following figure is a plot of the average efficiency of the different modulator orders with the 20 kHz data point removed from each order. While the 4th order modulator still adds lower average efficiency, it is now above 94.5%.

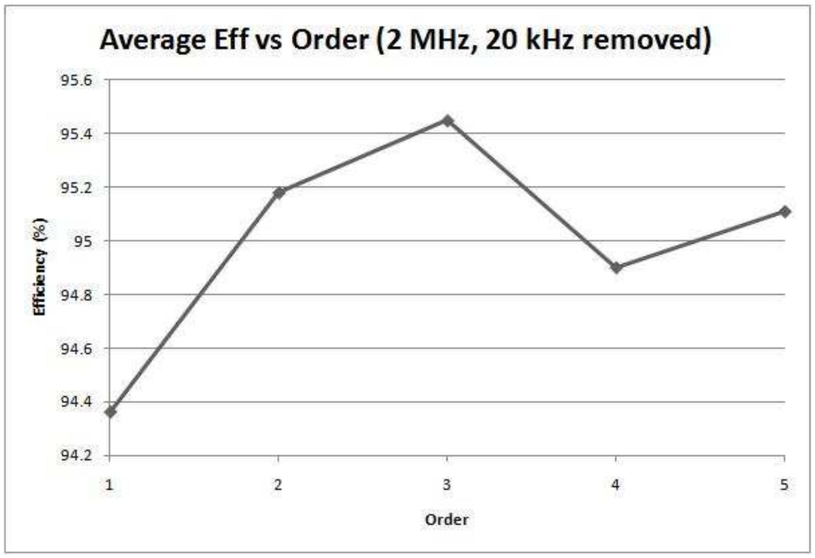


Figure 53 - Average efficiency vs. Modulator Order

With the removal of both 20 kHz and 12 kHz the 4th order modulator becomes one of the most efficient orders that was tested. The following plot shows the efficiency for only the 4th order. The plot shows consistently high efficiency until 5 kHz.

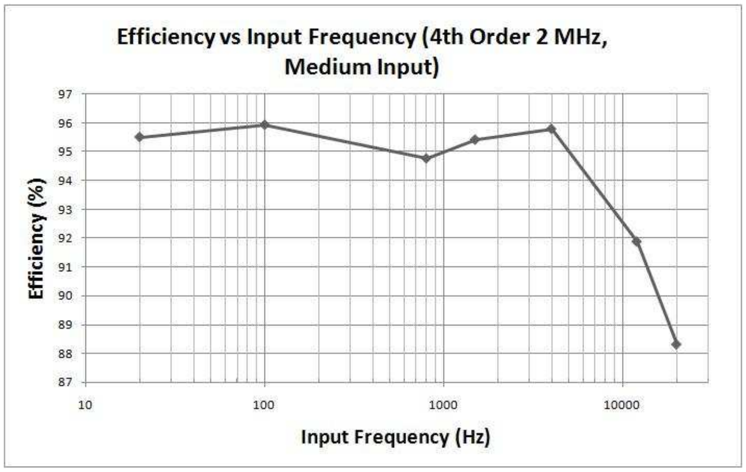


Figure 54 - Efficiency of a 4th Order Modulator

This final efficiency plot is the only one that has been displayed that has used an input amplitude of 2, previously described as a large input value. As seen from the power plots, higher input amplitudes result in higher output power. This also corresponds to higher power efficiency. The following plot shows efficiencies consistently above 97%.

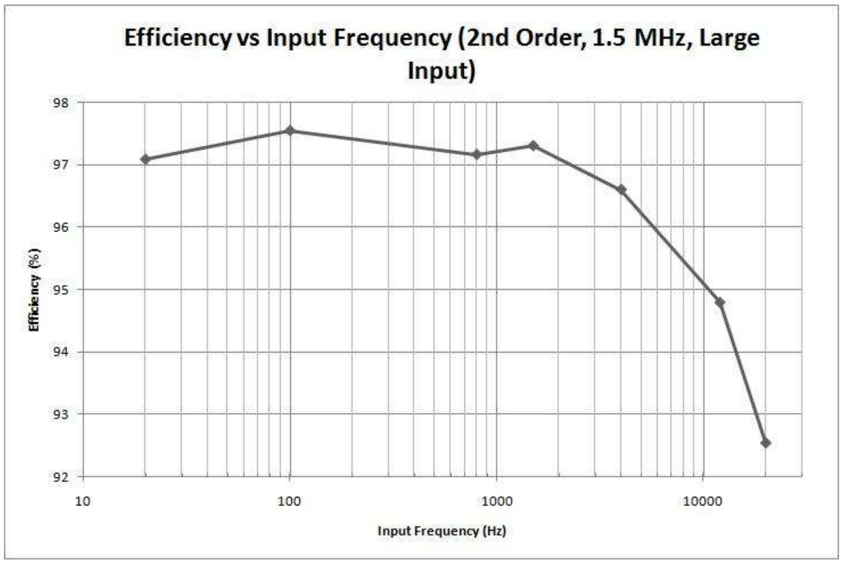


Figure 55 - Efficiency vs. Frequency

6 Conclusion

The initial goal of this MQP was to create a Class D amplifier that incorporated a digital input, full system feedback, over 100dB SNR and less than 0.5% THD. These goals required significant time and resources which would have made the task of designing an entirely new system too lengthy to complete in the normal MQP time frame. The previous year's Class D team created a solid power and filter design which we were able to reuse for our new design. Since the output power and efficiency are greatly dictated by the power stage our goals for these specifications were based off the results of the previous year.

Modulator design progressed mostly inside MATLAB. Through countless simulations the modulator was tweaked to create as high quality of an output as possible. Within MATLAB a virtual DSP was generated to aid in the testing of new modulator designs, and ease the transition from theoretical to practical. While MATLAB simulations were progressing the DSP was studied and its functionality explored in order to combine the simulation with the real device as seamlessly as possible.

The incorporation of feedback was greatly explored with in simulation. Many different configurations were tried, but no configuration was capable of producing any quality gains. While feedback simulations progress, the circuit required to capture the output and deliver it to the DSP was created. This circuit was designed onto a PCB along with the previous year's power and filter stage designs.

The DSP approach to the Class D amplifier proved to have many benefits over traditional configurations. In previous years teams have been forced to generate simulations, work out as many flaws as they can, decide on the best configuration of OSR and modulator order, and then construct their design in hope that there would be no errors. The DSP allowed many different modulator configurations to be applied to the same power stage and filter, allowing real life testing and comparison of limitless modulator configurations.

The DSP also aided in the successful implementation of a dual input system, allowing the amplifier to work with both analog and digital signals. With this configuration the design was able to amplify audio from simple devices like portable CD players, and more complicated devices like DVD players with coaxial digital audio connections.

The project also resulted in a catalog of useful MATLAB tools. These tools aided in the automated testing of many data points to determine SNR and THD numbers as quickly and accurately as possible, tools which could be reused in future years. It also resulted in code designed to automate the discovery of new and theoretical optimized coefficients for use in discrete time modulators.

This project helped redefine the audio quality requirements for the digital input Class D amplifier. Through careful examination it was concluded that traditional audio quality testing techniques, used for determining SNR, cannot be accurately applied to a digital input Class D modulator. As a result, MATLAB tools used to calculate SNR were designed to be robust in their ability to determine audio quality for specific inputs.

The final product of this MQP was a Class D audio amplifier that produced over 90 watts of output power, and was over 95% efficient. It was capable of over 110dB of 'zero-input' SNR, and over 50dB of standard SNR accepting both analog and SPDIF inputs.

7 Future Work and Recommendations

The Class D amplifier project runs nearly every year at WPI. Most years that the amplifier is developed the final report outlines recommendations to future Class D amplifier teams. One such recommendation was the incorporation of a digital input, which was implemented this year. It has also been suggested that full system feedback be added, which was attempted this year although it was not successful.

One of the most obvious, and commonly recommended additions is a high efficiency power supply for the Class D amplifier. The Class D amplifier requires a ripple free, high power DC supply. The cleanliness of the power supply rail directly affects the quality of the output. While a Class D amplifier is efficient, the conversion from AC to DC can be an extremely inefficient, negating the selling points of the Class D. Because the Class D amplifier requires a DC power supply, it has a perfect application in automotive audio. While an AC to DC power supply may be good, and DC to DC power supply capable of conditioning automotive power, which can fluctuate between 11 and 16 volts, would be a very good addition to the project.

With this year's addition of a digital input, the amplifier could be expanded to support multiple channels. DSP's have ample digital output pins that can be used to drive multiple power stages. SPDIF is capable of transmitting more than one channel. These channels could be extracted, separately modulated, and sent to their respective power stage.

It is common for consumer amplifiers to have signal processing features that provide the user with control over their listening experiences. Now that the Class D amplifier has been built into a DSP, a user interface could also be included to allow the user to adjust the EQ settings of the amplifier in order to compensate for low quality speakers or abnormal room conditions. Assuming the amplifier was servicing multiple channels it could be programmed to send high frequency to tweeters while sending low frequencies to mids or sub-woofers.

The output filter for the Class D amplifier requires the speaker to be the one resistor in the circuit. This has a few implications for the design. Variations in speakers, even among speakers rated as 8 ohm, will result in variations in their resistance. This will cause the filter to change cutoff frequency. The 8 ohm speaker is not the only available type. The 4 ohm and 6 ohm speakers are also common; these would drastically affect the filter cutoff. Speakers do not act as perfect passive devices. A speaker has a large coil and magnet used for pushing the cone. The speaker and magnet can have rippling affects that travel back through the amplifier and cause distortion. A good future addition would be the ability to support multiple speaker types, and incorporate circuit protection allowing the device to remain powered on even with no load attached, which could currently cause damage to the MOSFETS.

The Class D amplifier does not have to be relegated to the audio market. Class D amplifiers can be developed with a band-pass or high pass modulator, as opposed to a low pass modulator. The Class D amplifier can offer high efficiency amplification to devices in the RF region. A Class D amplifier could be configured as a band pass filter in the RF band and used to efficiently amplify digital and analog signals for communication or any long range wireless transmissions. It would be especially helpful in handheld devices which need to run on batteries.

References

1 Dorf, R. C. (1989). *Modern Control Systems*. Reading, Massachusetts: Addison-Welsley.

2 Kay, S. M. (1993). *Fundamentals of Statistical Signal Processing Estimation Theory Volume I*. Upper Saddle River, New Jersey: Prentice-Hall Inc.

3 Lathi, B. (2005). *Signals and Systems*. New York, New York: Oxford University Press.

4 Morey, B., Vasudevan, R., & Woloschin, I. (2008). *Class D Audio Amplifier*. Worcester, MA: WPI.

5 Schreier, R., & Temes, G. C. (2005). *Understanding Delta-Sigma Data Converters*. Hoboken, New Jersey: John Wiley & Sons, Inc.

6 Tomarakos, J. (2002, July 16). *The Relationship of Dynamic Range to Data Word Size in Digital Audio Processing*. Retrieved September 15, 2008, from http://www.audiodesignline.com/showarticle.jhtml?article=192200610

7 Wikipedia. (2008, September 2). Retrieved Spetember 8, 2008, from http://en.wikipedia.org/wiki/Pulse_density_modulation

Appendix A CIFB Modulator Coefficient Generation MATLAB Code

```
% In order to run this MATLAB script, the delta sigma toolbox of MATLAB
% must be installed
% This script was derived from the code fragment on pg. 285 of
% "Understanding Delta-Sigma Data Converters" by Richard Scgreier and Gabor
% C. Temes
function [a, g, b, c] = CalcCoefficients(order, OSR);
H = synthesizeNTF(order,OSR,0);
form = 'CIFB';
[a,g,b,c] = realizeNTF(H, form);
b(2:end) = 0;
ABCD = stuffABCD(a,g,b,c,form);
[ABCDs umax] = scaleABCD(ABCD);
[a, g, b, c] = mapABCD(ABCDs,form)
```

Appendix B Loop Filter Coefficient Generation MATLAB Code

```
function [NTFtop, NTFbottom, Gbottom, Gtop, Hbottom, Htop] =
SynthesizePlus(order)
[NTF, z, p] = MysynthesizeNTF(order);
NTF.variable ='z^-1';
NTF = tf(NTF);
G = 1/NTF;
H = 1 - NTF;
NTFtop = poly(z);
NTFbottom = poly(p);
Gtop = NTFbottom;
Gbottom = NTFtop;
Htop = NTFbottom - NTFtop;
Hbottom = Gtop;

function[ntf, z, p] = MysynthesizeNTF(order,osr,opt,H_inf,f0)
%This function is a modified version of synthesizeNTF, which is included in
%the Delta-Sigma toolbox. Many thanks to the creater of the toolbox,
%Richard Schreier.
%ntf = synthesizeNTF(order=3,osr=64,opt=0,H_inf=1.5,f0=0)
%Synthesize a noise transfer function for a delta-sigma modulator.
% order = order of the modulator
% osr = oversampling ratio
% opt = flag for optimized zeros
% 0 -> not optimized,
% 1 -> pre-computed optima (good for high osr)
% 2 -> as above with at least one zero at band-center
% 3 -> optimized zeros (Requires MATLAB6 and Optimization Toolbox)
% [] -> zero locations in complex form
% H_inf = maximum NTF gain
% f0 = center frequency (1->fs)
%
%ntf is a zpk object containing the zeros and poles of the NTF. See zpk.m
%
% See also
% clans() "Closed-loop analysis of noise-shaper." An alternative
% method for selecting NTFs based on the 1-norm of the
% impulse response of the NTF
%
% synthesizeChebyshevNTF() Select a type-2 highpass Chebyshev NTF.
% This function does a better job than synthesizeNTF when order
% is high and H_inf is low.

% Handle the input arguments
parameters = {

'order' 'osr' 'opt' 'H_inf' 'f0'};
defaults = { 3 64 0 1.5 0 };
for
arg_i=1:length(defaults)
parameter = char(parameters(arg_i));
if arg_i>nargin | ( eval(['isnumeric(' parameter ') '])
eval([
'any(isnan(' parameter ')) | isempty(' parameter ') ']) )
eval([parameter '=defaults{arg_i};'])
end
end
if f0 > 0.5
fprintf(1, 'Error. f0 must be less than 0.5.\n');
return;
end
if f0 ~= 0 & f0 < 0.25/osr
warning('(%s) Creating a lowpass ntf.', mfilename);
f0 = 0;
end

if f0 ~= 0 & rem(order,2) ~= 0
fprintf(1,'Error. order must be even for a bandpass modulator.\n');
return;
end

if length(opt)>1 & length(opt)~=order
fprintf(1,'The opt vector must be of length %d(=order).\n', order);
return;
end

% Determine the zeros.
if f0~=0 % Bandpass design-- halve the order temporarily.
order = order/2;
```

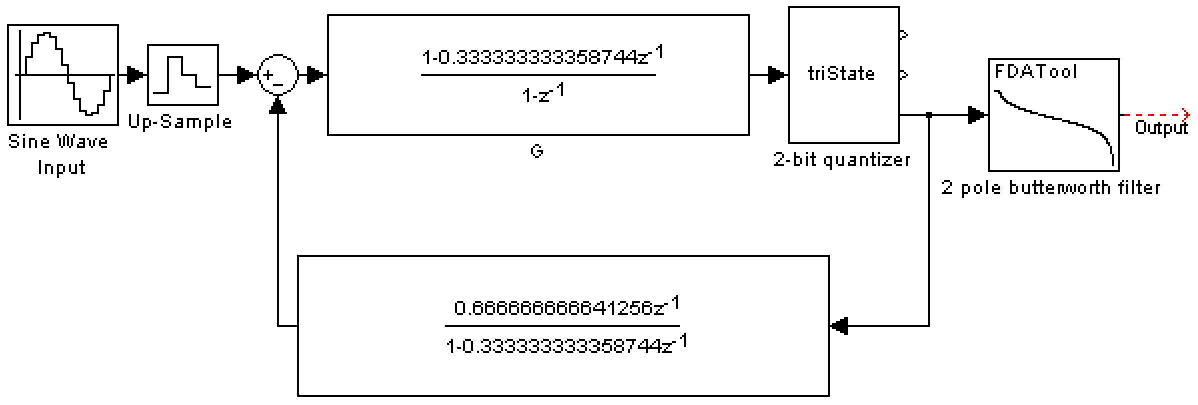
```
dw = pi/(2*osr);
else
dw = pi/osr;
end

if length(opt)==1
if opt==0
z = zeros(order,1);
else
z = dw*ds_optzeros(order,1+rem(opt-1,2));
if isempty(z)
return;
end
end

if f0~=0 % Bandpass design-- shift and replicate the zeros.
order = order*2;
z = z + 2*pi*f0;
ztmp = [ z'; -z' ];
z = ztmp(:);
end
z = exp(j*z);
else
z = opt(:);
end

ntf = zpk(z,zeros(1,order),1,1);
Hinf_itn_limit = 100;

opt_iteration = 5;

% Max number of zero-optimizing/Hinf iterations
while opt_iteration > 0

% Iteratively determine the poles by finding the value of the x-parameter
% which results in the desired H_inf.
ftol = 1e-10;

if f0>0.25
z_inf=1;
else
z_inf=-1;
end

if f0 == 0 % Lowpass design
HinfLimit = 2^order;

% !!! The limit is actually lower for opt=1 and low osr
if H_inf >= HinfLimit
fprintf(2,'%s warning: Unable to achieve specified Hinf.\n', mfilename);
fprintf(2,'Setting all NTF poles to zero.\n');
ntf.p = zeros(order,1);

else
x=0.3^(order-1);

% starting guess
```

```
converged = 0;

for itn=1:Hinf_itn_limit
me2 = -0.5*(x^(2./order));
w = (2*[1:order]'-1)*pi/order;
mb2 = 1+me2*exp(j*w);
p = mb2 - sqrt(mb2.^2-1);
out = find(abs(p)>1);
p(out) = 1./p(out);

% reflect poles to be inside the unit circle.
p = cplxpair(p);
ntf.z = z; ntf.p = p;
f = real(evalTF(ntf,z_inf))-H_inf;


% [ x f ]
if itn==1
delta_x = -f/100;
else
delta_x = -f*delta_x/(f-fprev);
end

xplus = x+delta_x;

if xplus>0
x = xplus;
else
x = x*0.1;
&nbsp;
end

fprev = f;

if abs(f)<ftol | abs(delta_x)<1e-10
converged = 1;
break;
end

if x>1e6
fprintf(2, '%s warning: Unable to achieve specified Hinf.\n', mfilename);
fprintf(2, 'Setting all NTF poles to zero.\n');
ntf.z = z; ntf.p = zeros(order,1);
break;
end

if itn == Hinf_itn_limit
fprintf(2,'%s warning: Danger! Iteration limit exceeded.\n', mfilename);
end
end
end
else % Bandpass design.
x = 0.3^(order/2-1);
% starting guess (not very good for f0~0)
c2pif0 = cos(2*pi*f0);

for itn=1:Hinf_itn_limit
```

```
e2 = 0.5*x^(2./order);
w = (2*[1:order]'-1)*pi/order;
mb2 = c2pif0 + e2*exp(j*w);
p = mb2 - sqrt(mb2.^2-1);


% reflect poles to be inside the unit circle.
out = find(abs(p)>1);
p(out) = 1./p(out);
p = cplxpair(p);
ntf.z = z; ntf.p = p;
f = real(evalTF(ntf,z_inf))-H_inf;

% [x f]
if itn==1
delta_x = -f/100;
else
delta_x = -f*delta_x/(f-fprev);
end

xplus = x+delta_x;

if xplus > 0
x = xplus;
else
x = x*0.1;
end
fprev = f;

if abs(f)<ftol | abs(delta_x)<1e-10
break;
end
if x>1e6
fprintf(2,'%s warning: Unable to achieve specified Hinf.\n', mfilename);
fprintf(2,'Setting all NTF poles to zero.\n');
p = zeros(order,1);
ntf.p = p;
break;
end
if itn == Hinf_itn_limit
fprintf(2,'%s warning: Danger! Hinf iteration limit exceeded.\n', mfilename);
end
end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if opt < 3 % Do not optimize the zeros
opt_iteration = 0;
else
zp = z(angle(z)>0);
x0 = (angle(zp)-2*pi*f0) * osr / pi;

if opt==4 & f0~=0

% Do not optimize the zeros at f0
x0(find( abs(x0)<1e-10 )) = [];
end
```

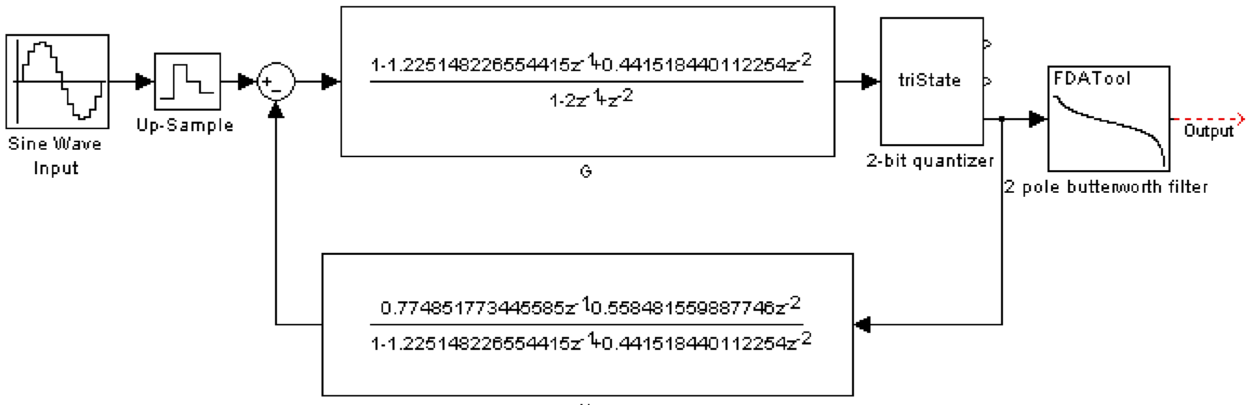
```
if f0 == 0
ub = ones(size(x0));
lb = zeros(size(x0));


else
ub = 0.5*ones(size(x0));
lb = -ub;
end

options = optimset(

'TolX',0.001, 'TolFun',0.01, 'MaxIter',100 );
options = optimset(options, 'LargeScale','off');
options = optimset(options, 'Display','off');

%options = optimset(options,'Display','iter');
x = fmincon(@(x)ds_synNTFobj1(x,p,osr,f0),x0,[],[],[],[], lb,ub,[],options);
z = exp(2i*pi*(f0+0.5/osr*x));

if f0>0
z = padt(z,length(p)/2,exp(2i*pi*f0));
end

z = [z conj(z)]; z = z(:);

if f0==0
z = padt(z,length(p),1);
end

ntf.z = z; ntf.p = p;

if abs( real(evalTF(ntf,z_inf)) - H_inf ) < ftol
opt_iteration = 0;
else
opt_iteration = opt_iteration - 1;
end
end
end
```

Appendix C Loop Filter Simulink Models



Sine Wave Input
Up-Sample
+
-
$\frac{1-0.333333333358744z^{-1}}{1-z^{-1}}$
G
triState
2-bit quantizer
FDATool
2 pole butterworth filter
Output
$\frac{0.666666666641256z^{-1}}{1-0.333333333358744z^{-1}}$
H

Figure 56 - First Order Loop Filter Simulink Model



Sine Wave Input
Up-Sample
+
-
$\frac{1-1.225148226554415z^{-1}+0.441518440112254z^{-2}}{1-2z^{-1}+z^{-2}}$
G
triState
2-bit quantizer
FDATool
2 pole butterworth filter
Output
$\frac{0.774851773445585z^{-1}-0.558481559887746z^{-2}}{1-1.225148226554415z^{-1}+0.441518440112254z^{-2}}$
H

Figure 57 - Second Order Loop Filter Simulink Model

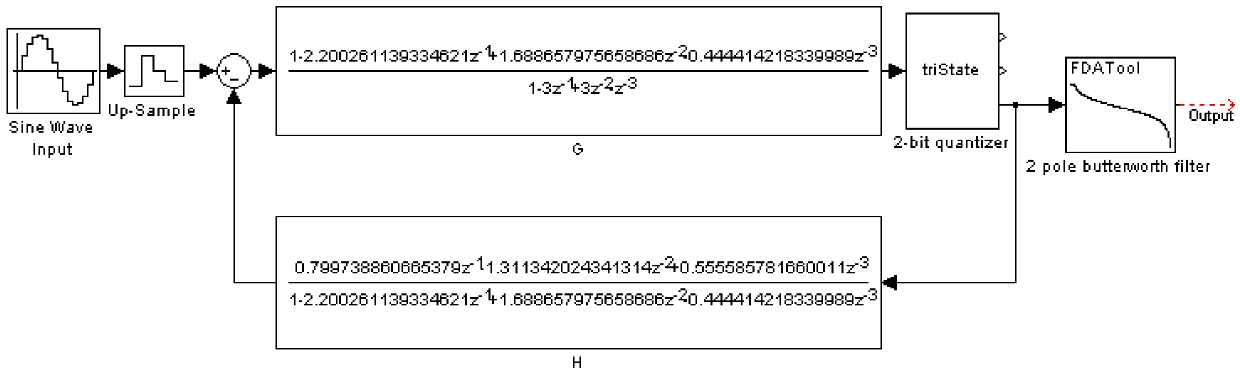

Sine Wave Input
Up-Sample
+
-
1-2.200261139334621z-1+1.688657975658686z-2 0.444414218339989z-3
1-3z-1+3z-2 z-3
G
triState
2-bit quantizer
FDATool
Output
2 pole butterworth filter
0.799738860665379z-1 1.311342024341314z-2+0.555585781660011z-3
1-2.200261139334621z-1+1.688657975658686z-2 0.444414218339989z-3
H

Figure 58 - Third Order Loop Filter Simulink Model

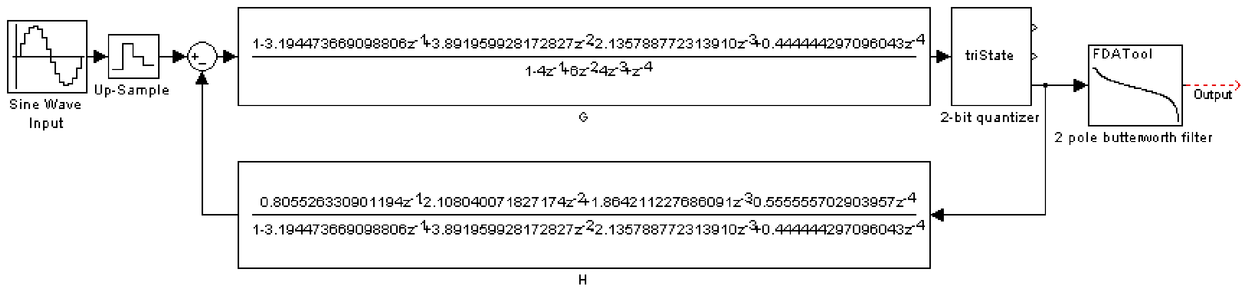

Sine Wave Input
Up-Sample
+
-
1-3.194473669098806z-1+3.891959928172827z-2 2.135788772313910z-3+0.444444297096043z-4
1-4z-1+6z-2 4z-3+z-4
G
triState
2-bit quantizer
FDATool
Output
2 pole butterworth filter
0.805526330901194z-1 2.108040071827174z-2+1.864211227686091z-3 0.555555702903957z-4
1-3.194473669098806z-1+3.891959928172827z-2 2.135788772313910z-3+0.444444297096043z-4
H

Figure 59 - Forth Order Loop Filter Simulink Model

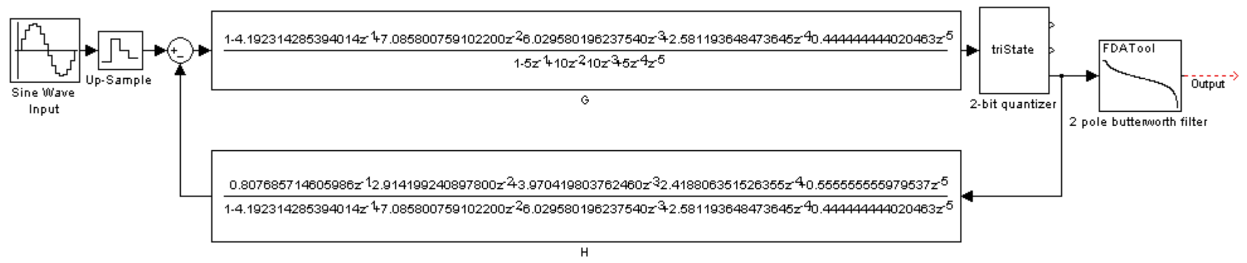

Sine Wave Input
Up-Sample
+
-
1-4.192314285394014z-1+7.085800759102200z-2 6.029580196237540z-3+2.581193648473645z-4 0.444444444020463z-5
1-5z-1+10z-2 10z-3+5z-4 z-5
G
triState
2-bit quantizer
FDATool
Output
2 pole butterworth filter
0.807685714605986z-1 2.914199240897800z-2+3.970419803762460z-3 2.418806351526355z-4+0.555555555979537z-5
1-4.192314285394014z-1+7.085800759102200z-2 6.029580196237540z-3+2.581193648473645z-4 0.444444444020463z-5
H

Figure 60 - Fifth Order Loop Filter Simulink Model

Appendix D DSP Simulation MATLAB Code

```
function [ input, output ] = DSPsim(f,fs,dur,ord,amp)

global order thrsh gain a_H b_H a_G b_G u_G u_H h s

h  = fdesign.lowpass('N,Fc', 2, 30000, 2000000);
lpf= design(h, 'butter');

order = ord;
thrsh = 0.5;
gain = 2;

%Fifth order
if(order==5)
    a_G = [1, -5, 10, -10, 5, -1];
    b_G = [1, -4.192314285394014, 7.085800759102200, -6.029580196237540,
2.581193648473645, -0.444444444020463];

    a_H = [1, -4.192314285394014, 7.085800759102200, -6.029580196237540,
2.581193648473645, -0.444444444020463];
    b_H = [0.807685714605986, -2.914199240897800, 3.970419803762460, -
2.418806351526355, 0.555555555979537, 0];

%Fourth order
elseif(order==4)
    a_G = [1, -4, 6, -4, 1];
    b_G = [1, -3.194473669098806, 3.891959928172827, -2.135788772313910,
0.444444297096043];

    a_H = [1, -3.194473669098806, 3.891959928172827, -2.135788772313910,
0.444444297096043];
    b_H = [0.805526330901194, -2.108040071827174, 1.864211227686091, -
0.555555702903957, 0];

 %Third order
elseif(order==3)

    a_H = [1 -2.199 1.687 -0.4439];
    b_H = [0.7985 -1.31 0.5561 0];

    a_G = [1 -2.998 2.998 -1];
    b_G = [1 -2.199 1.687 -0.4439];



%Second order
elseif(order==2)
     a_H = [1 -1.225 0.4413];
     b_H = [0.7774 -0.5587 0];
     a_G = [1 -1.999 1];
     b_G = [1 -1.225 0.4413];
end

%Create input signal
t=linspace(0,dur,dur*fs);
inBuf=amp*sin(2*pi*f*t);

%Initialize variables
u_G= zeros(1,order+1,'double');
u_H= zeros(1,order+1,'double');

h = 0;
s = 0;

sz=length(inBuf);
outBuf=zeros(1,sz,'double');

%Modulate
for i=1:1:length(inBuf)
    outBuf(i)=modulate(inBuf(i));
end

%Filter
lowpassed=filter(lpf,outBuf);

%plot
subplot(4,1,1), plot(t,inBuf);
subplot(4,1,2), plot(t,outBuf);
subplot(4,1,3), plot(t,lowpassed);
subplot(4,1,4),
semilogx(linspace(0,fs,length(t)),20*log10(abs(fft(lowpassed))));

%Generate signals for calcSNR
input.signals.values=inBuf;
input.time=t;
output.signals.values=lowpassed;
output.time=t;
end

function [y] = modulate(in)
    global order a_H b_H a_G b_G h s
    s=in-h;
    g=iir(s,a_G,b_G,0,order);
    y=quantize(g);
    h=iir(y,a_H,b_H,1,order);
end


function [y] = iir(in,a,b,buffer,order)
    y=0;
    global u_G u_H
    if(buffer==0) %u_G
```

```
        %circulate buffer
        for i=order+1:-1:2
            u_G(i)=u_G(i-1);
        end
        u_G(1)=in*a(1);

        for i=2:1:order+1
            u_G(1)= u_G(1) - (a(i)*u_G(i));
        end
        for i=1:1:order+1
            y=y+(b(i)*u_G(i));
        end
    else
        %circulate buffer
        for i=order+1:-1:2
            u_H(i)=u_H(i-1);
        end
        u_H(1)=in*a(1);

        for i=2:1:order+1
            u_H(1)= u_H(1) - (a(i)*u_H(i));
        end
        for i=1:1:order+1
            y=y+(b(i)*u_H(i));
        end
    end
end

function [y] = quantize(in)
    global thrsh gain
    if(in<-thrsh)
        y=-gain;
    elseif(in>thrsh)
        y=gain;
    else
        y=0.0;
    end
end
```

Appendix E Coefficient Optimization MATLAB Code

```
function [snr] = estimateCoeffs(coeffs)
fs=2000000; %sampling frequency of 2Mhz
dur=0.01;  %duration
```

%Parse the coeffs matrix into individual arrays of coefficients

```
a_H = coeffs(1,:);
b_H = coeffs(2,:);
a_G = coeffs(3,:);
b_G = coeffs(4,:);

%Determine the order of the given coefficients
order=length(a_H)-1;

%Create a low pass filter for the output of the modulator
h  = fdesign.lowpass('N,Fc', 2, 30000, 2000000);
Hd = design(h, 'butter');

%Generate a set of 20 frequencies between 1kHz and 20kHz
f=linspace(1000,20000,20);

%For each frequency, do DSP simulation, and calculate the SNR of the output
for i=1:1:length(f)
    [input, output] = DSPsimCoeffs(f(i),fs,dur,Hd, order, a_H,b_H,a_G,b_G);
    snrs(i)=newestCalcSNR(output,f(i),fs,0);
end

%Take the average of all snr's and return it negated (needed for fminsearch)
snr=-(sum(snrs)/length(f));



function [result] = justdoit(mult)

tic;
for i=1:1:5
    order=i;
      %Fifth order
    if(order==5)
        a_G = [1, -5, 10, -10, 5, -1];
        b_G = [1, -4.192314285394014, 7.085800759102200, -6.029580196237540,
                  2.581193648473645, -0.444444444020463];

        a_H = [1, -4.192314285394014, 7.085800759102200, -6.029580196237540,
                  2.581193648473645, -0.444444444020463];
        b_H = [0.807685714605986, -2.914199240897800, 3.970419803762460,
                  -2.418806351526355, 0.555555555979537, 0];

    %Fourth order
    elseif(order==4)
        a_G = [1, -4, 6, -4, 1];
        b_G = [1, -3.194473669098806, 3.891959928172827, -2.135788772313910,
0.444444297096043];

        a_H = [1, -3.194473669098806, 3.891959928172827, -2.135788772313910,
0.444444297096043];
        b_H = [0.805526330901194, -2.108040071827174, 1.864211227686091, -
0.555555702903957, 0];

    %Thir order
    elseif(order==3)

        a_H = [1 -2.200261139334621 1.688657975658686 -0.444414218339989];
        b_H = [0.799738860665379 -1.311342024341314 0.555585781660011 0];

        a_G = [1 -3 3 -1];
        b_G = [1 -2.200261139334621 1.688657975658686 -0.444414218339989];


    %Second order
    elseif(order==2)
         a_H = [1 -1.225148226554415 0.441518440112254];
         b_H = [0.774851773445585 -0.558481559887746 0];

         a_G = [1 -2 1];
         b_G = [1 -1.225148226554415 0.441518440112254];

    %First order
    elseif (order==1)
        a_H = [1.000000000000000  -0.333333333358744];
        b_H = [0.666666666641256 , 0];

        a_G=[1 -1];
        b_G=[1.000000000000000  -0.333333333358744];
    end
    %Do the magic
    result(i).order='Order'+i;
    result(i).oldcoeffs=[a_H;b_H;a_G;b_G];
    result(i).oldsnr = -estimateCoeffs(result(i).oldcoeffs);
    iter=mult*(4+i*4);
    options = optimset('MaxIter',iter,'MaxFunEvals',iter);
    [result(i).coefs,snr] =
fminsearch(@estimateCoeffs,[a_H;b_H;a_G;b_G],options);
    result(i).snr=-snr;
end

toc

for i=1:1:5
    x(i)=i;
    y1(i)=result(i).oldsnr;
    y2(i)=result(i).snr;
end

plot(x,y1,'--rs','LineWidth',2,...
            'MarkerEdgeColor','k',...
            'MarkerFaceColor','g',...
            'MarkerSize',10);
hold on;
plot(x,y2,'--rs','LineWidth',2,...
            'MarkerEdgeColor','b',...
            'MarkerFaceColor','r',...
            'MarkerSize',10);
hold off;
```

Appendix F Automated Testing MATLAB Code

getStats.m

```
function [snr,freq,thd,time] = getStats(fs)
    for j=1:1:3
        for i=1:1:5
           for n=1:1:18
                [signal]=readFile(j,i,n);
                thd(j,i,n)=THD(signal,fs);
                tic;
                [snr(j,i,n),freq(j,i,n)] = optSNR(signal,fs);
                time(j,i,n)=toc;

                out=['SNR: ',num2str(snr(j,i,n))]
                out=['FREQ: ',num2str(freq(j,i,n))]
                out=['THD: ',num2str(thd(j,i,n))]
                out=['TIME: ',num2str(time(j,i,n))]

            end
        end
    end
end

function [signal] = readFile(j,i,n)
    switch j
        case 1
            modFreq = '1MHz';
        case 2
            modFreq = '15MHz';
        case 3
            modFreq = '2MHz';
    end
    name = ['.\cut_files\', modFreq, num2str((i-1)*19+n+1),'.wav'];
    %name=['Ord ',num2str(i),' - ',num2str(n),'.wav']
    signal = wavread(name);
end

function [thd1] = THD(output,fs)
    %freqs = fft(output.signals.values);
    y=output;
    L=length(y);
    NFFT = 2^nextpow2(L); % Next power of 2 from length of y
    Y = fft(y,NFFT)/L;
    f = fs/2*linspace(0,1,NFFT/2);
    % Plot single-sided amplitude spectrum.
    plot(f,2*abs(Y(1:NFFT/2)))
    title('Single-Sided Amplitude Spectrum of y(t)')
    xlabel('Frequency (Hz)')
    ylabel('|Y(f)|')

    %Number of harmonics
    harms = 5;
    Y=abs(Y);
    %Find index of fundamental frequency
    [temp,index]=max(Y);
    %Get magnitude of fundamental freq.
    fFund = Y(index);
    %Get magnitude of first 5 harmonics
    h1 = Y(2*index)^2;
    h2 = Y(3*index)^2;
    h3 = Y(4*index)^2;
    h4 = Y(5*index)^2;
    h5 = Y(6*index)^2;
    %Calculate their sum
    hSum = sqrt(h1+h2+h3+h4+h5);
    %Calculate THD
    thd1=(hSum/fFund)*100;
end
```

optSNR.m

```
function [snr,f]= optSNR(output,fs)
global x;
global fs1;

fs1=fs;
x= output;

f_opt = optF(output);
[f,snr] = fminsearch(@calcSNR,f_opt);
snr = -snr;

end

function f_opt = optF(output)
    global fs1;
    fDomain = abs(fft(output));
    [c,i] = max(fDomain);
    f_opt = fs1*i/(length(fDomain)-1);
end

function [snr] = calcSNR(f)
global x;
global fs1;
fs = fs1;
r = 1; % A multiplyier that controls how much space is between truncations
for b = 1:1:8*fs/f
    y = x((r*b):length(x));
    n = [0:length(y)-1]';
    A_hat=abs(y'*exp(-1j*2*pi*f*n/fs))/length(y)*2;
    theta_hat=atan(-(y'*sin(2*pi*f*n/fs)) / (y'*cos(2*pi*f*n/fs)));
    x_hat=A_hat*cos(2*pi*f*n/fs+theta_hat);
    % Time Domain Method
    w=y-x_hat;
    biggest(b) = max(w);
end
[smallest, i] = min(biggest);
i = i(1);
   y = x((r*i):length(x));
    n = (0:length(y)-1)';
    A_hat=abs(y'*exp(-1j*2*pi*f*n/fs))/length(y)*2;
    theta_hat=atan(-(y'*sin(2*pi*f*n/fs)) / (y'*cos(2*pi*f*n/fs)));
    x_hat=A_hat*cos(2*pi*f*n/fs+theta_hat);

    % Time Domain Method
    w=y-x_hat;
    w1 = fft(w);
    %figure(1);
    plot(w);
    N1 = length(w1);
    wtrunc = w1(1:floor((length(w1)/fs)*20000));
    sigPWR1 = x_hat'*x_hat;
    noisePWR1 = (wtrunc'*wtrunc)/(N1+1);
    snr = -10*log10(sigPWR1/noisePWR1);
end
```

getZeroSNR.m

```
function [snr,freq,thd,time] = getZeroSNR(fs)
for k=1:1:3
    for j=1:1:3
        for i=1:1:5
            for n=1:1:19
                [signal,noise]=readFile(k,j,i,n);
                [snr(k,j,i,n),freq(k,j,i,n)] = getSNR(signal,fs);
                time(k,j,i,n)=toc;

                out=['SNR: ',num2str(snr(k,j,i,n))]
                out=['FREQ: ',num2str(freq(k,j,i,n))]
                out=['THD: ',num2str(thd(k,j,i,n))]
                out=['TIME: ',num2str(time(k,j,i,n))]

            end
        end
    end
end

end

function [signal,noise] = readFile(k,j,i,n)
    switch j
        case 1
            modFreq = '1MHz';
        case 2
            modFreq = '15MHz';
        case 3
            modFreq = '2MHz';
    end
    signalName = ['.\cut_files\', modFreq, num2str((i-1)*19+n),'.wav'];
    noiseName = ['.\zero input\',num2str(k),'.wav'];
    %name=['Ord ',num2str(i),' - ',num2str(n),'.wav']
    signal = wavread(signalName);
    noise = wavread(noiseName);
end


function [snr,freq]= getSNR(signal,noise,fs)
    sig = abs(fft(signal));
    freq = max(sig);
    sigPwr = max(sig)^2;
    noisePwr = sum(noise.^2);
    snr = 10*log10(sigPwr/noisePwr);
end
```

format.m

```
function [page] = format(var)
page = zeros(5*3,18);
row=1;
for i=1:1:3
    for j=1:1:5
        temp = var(i,j,:);
        page(row,:) = temp(:)';
        row=row+1;
    end
end
```

plotForUs.m

```
function plotForUs(var,xPlot,yPlot)
    [x, xLabel]=getData(var,xPlot);
    [y, yLabel]=getData(var,yPlot);


    subplot(3,1,1), myPlot(x,y,1,5);
    title('1Mhz Modulation','fontweight','b');
    xlabel(xLabel);
    ylabel(yLabel);
    subplot(3,1,2), myPlot(x,y,6,10);
    title('1.5Mhz Modulation','fontweight','b');
    xlabel(xLabel);
    ylabel(yLabel);
    subplot(3,1,3), myPlot(x,y,11,15);
    title('2Mhz Modulation','fontweight','b');
    xlabel(xLabel);
    ylabel(yLabel);


end
```

```
function myPlot(x,y,first,last)

    for i=first:1:last
        plot(x(i,:),y(i,:),'-','Color',getColor(i),'LineWidth',0.5);
        hold on;
    end
    hold off;
    legend('1st Order','2nd Order','3rd Order','4th Order','5th Order');
    legend('show');
    grid on;

end

function [color]= getColor(i)
    switch mod(i,5)

        case 1
            color = [0.051 0.733 0.18];
        case 2
            color = [206/255 36/255 8/255];
        case 3
            color = [32/255 123/255 224/255];
        case 4
            color = [153/255 82/255 232/255];
        case 0
            color = [213/255 132/255 13/255];
    end
end

function [out,label] = getData(var,plotVar)
    switch plotVar
        case 'snr'
            label='SNR (dB)';
            out = var.snr;
        case 'freq'
            label = 'Frequency (Hz)';
            out = var.freq;
        case 'thd'
            label = 'THD (%)';
            out = var.thd;
        case 'time'
            label = 'Calc Time (s)';
            out = var.time;
    end
end
```

Appendix G DSP Code

Main.c

```
/**********************************************
*
*     Class D Amplifier 2009
*
***********************************************/

#include "tt.h"
#include <stdio.h>
#include <math.h>
#include <SRU.h>
#include <time.h>
#include <sysreg.h>


#define orders 5  //Defines the modulator order
#define freqs 20  //used for testing 20 different frequencies
#define amp 1.5   //quantizer gain
#define N   296   //timer max count (Used for Timer ISR)

#define G 0                   //filter types
#define H 1

#define min -2    // Min Output of quantizer
#define max 2// Max output of quantizer
#define thr_Lo    -0.5// Threshold For Quantizer
#define thr_Hi    0.5// Threshold for Quantizer
#define time_diff 1.0416666666666666e-5   //Time increment for 96kHz
#define time_diff2 0.0000005   //Time increment for 2 Mhz
#define PI 3.14159265358979323846


/*Function declarations*/
double iir(double iirin, const double *a, const double *b, double *u);
double quantize(double inp, double low, double high);
void modulate(void);
void clk(void);
void initVars(void);
void updateCoeffs(void);
void doTriState(int status);

/* Global variables */
int   in,in2;
bool flag,flag1,flag2;
double x,x1,x2,x3,x4,s,h,g,y, adc, spdif;
double a_H[6],b_H[6],a_G[6],b_G[6];
double freq[20];
```

```
int order,orderPrev;
double timeADC, timeFast;
double u_H[6];    //H filter U buffer
double u_G[6];    //G filter U buffer
double adc_buf[2048],spdif_buf[1024];
int count;
int freqNum;

/*timer variables*/
volatile clock_t clock_start;
volatile clock_t clock_stop;
double secs,outf;
int triState;

void main(void)
{

    //Initialize PLL to run at CCLK= 331.776 MHz & SDCLK= 165.888 MHz
    InitPLL_SDRAM();
    // Need to initialize DAI because the sport signals need to be routed
    InitSPDIF();
    // This function will configure the codec on the kit
    Init1835viaSPI();
    // Finally setup the sport to receive / transmit the data
    InitSPORT();


    order=orders;
    orderPrev=orders;  //used for testing
    updateCoeffs();     //select coefficients based on order
    freqNum=0;          //used for testing

    initVars();         //initialize all variables

    /*Create 20 different frequencies, starting a 20Hz, and incrementing by
    1k 20 times.  Used for testing only*/
    freq[0]=20;
    int i;
    for(i=1;i<20;i++)
    {
      freq[i]=freq[i-1]+1000;
    }

    /*Setup input ISR (choose either adc or spdif)*/
    //interruptf(SIG_SP0,adcISR);
    interruptf(SIG_SP0,spdifISR);
    interruptf(SIG_TMZ0, timerISR); //enable high priority timer interrupt

      timer_set(N, N);              //set tperiod and tcount of the timer
      timer_on();                   //start timer


      while(1){
            //Loop forever, interrupts will occur while this loop runs
      }
}
```

```
void initVars()
{
    //initialize variables
    h=0.0;
    adc=0.0;
    s=0.0;
    g=0.0;
    y=0.0;
    count=0;
    flag2=0;
    flag1=0;
    flag=0;
    timeADC=0;
    outf=0;
    triState=2;

    int i;
    for (i=0;i< 6;i++)
    {
      u_H[i]=0.0;
      u_G[i]=0.0;
    }
}

void updateCoeffs(void)
/*This function is used to choose the coefficients to use based on the
order*/
{
      if(order==1)
      {
            double a_Hs[6]=  { 1.050000000000000,   -0.333333333358744,
                              0,0,0,0};
            double b_Hs[6]=  { 0.666666666641256,
                              0,0,0,0,0};
            double a_Gs[6]=  {1.000000000000000,    -1.000000000000000,
                              0,0,0,0};
            double b_Gs[6]=  {1.000000000000000,    -0.333333333358744,
                              0,0,0,0};
            int i=0;
            for(i;i<6;i++)
            {
                  a_H[i]=a_Hs[i];
                  b_H[i]=b_Hs[i];
                  a_G[i]=a_Gs[i];
                  b_G[i]=b_Gs[i];
            }

      }
            else if(order==2)
            {
                  double a_Hs[6] = {1.000000000000000,  -1.225148226554415,
                                    0.441518440112254,0,0,0};
                  double b_Hs[6] = {0.774851773445585,  -0.558481559887746,
                                    0,0,0,0};
                  double a_Gs[6] = {1.000000000000000,  -2.000000000000000,
                                    1.000000000000000,0,0,0};
                  double b_Gs[6] = { 1.050000000000000,  -1.225148226554415,
                                  0.441518440112254,0,0,0};
                  int i=0;
                  for(i;i<6;i++)
                  {
                        a_H[i]=a_Hs[i];
                        b_H[i]=b_Hs[i];
                        a_G[i]=a_Gs[i];
                        b_G[i]=b_Gs[i];
                  }
            }
            else if(order==3)
            {
                  double a_Hs[6] = {1,-2.2004,  1.6887514,-0.44441466};
                  double b_Hs[6] = {0.79974,-1.31157,   0.5556, 0};

                  double a_Gs[6] = {1,-3,3,-1};
                  double b_Gs[6] = {1,-2.2004,1.6887514,-0.44441466};
                  int i=0;
                  for(i;i<6;i++)
                  {
                        a_H[i]=a_Hs[i];
                        b_H[i]=b_Hs[i];
                        a_G[i]=a_Gs[i];
                        b_G[i]=b_Gs[i];
                  }
            }
            else if(order==4)
            {
                  double a_Hs[6]={1.000000000000000,  -3.194473669098806,
                                  3.891959928172827,  -2.135788772313910,
                                  0.444444297096043,0};
                  double b_Hs[6]={0.805526330901194,  -2.108040071827174,
                                  1.864211227686091,  -0.555555702903957,
                                  0,0};
                  double a_Gs[6]={1.050000000000000,  -4.000000000000000,
                                  6.000000000000000,  -4.000000000000000,
                                  1.000000000000000,0};
                  double b_Gs[6]={1.000000000000000,  -3.194473669098806,
                                  3.891959928172827,  -2.135788772313910,
                                  0.444444297096043,0};
                  int i=0;
                  for(i;i<6;i++)
                  {
                        a_H[i]=a_Hs[i];
                        b_H[i]=b_Hs[i];
                        a_G[i]=a_Gs[i];
                        b_G[i]=b_Gs[i];
                  }
            }
            else if(order==5)
            {
                  double a_Hs[6]= {1.000000000000000,  -4.192314285394014,
                                  7.085800759102200,  -6.029580196237540,
                                  2.581193648473645,  -0.444444444020463};
                  double b_Hs[6]= {0.807685714605986,  -2.914199240897800,
                                  3.970419803762460,  -2.418806351526355,
                                  0.555555555979537,                   0};
                  double a_Gs[6]= {1.000000000000000,  -5.000000000000000,
                                  10.000000000000000, -10.000000000000000,
                                  5.000000000000000,  -1.000000000000000};
                  double b_Gs[6]= {1.000000000000000,  -4.192314285394014,
                                  7.085800759102200,  -6.029580196237540,
                                  2.581193648473645,  -0.444444444020463};
                  int i=0;
                  for(i;i<6;i++)
                  {
                        a_H[i]=a_Hs[i];
                        b_H[i]=b_Hs[i];
                        a_G[i]=a_Gs[i];
                        b_G[i]=b_Gs[i];
                  }
            }
}

/***********************************************
*
*     ISR for ADC Input
*
***********************************************/
void adcISR(int sig)

{
    while((*pSPCTL0>>30)!=3)
    {}
      in2 = *pRXSP0A;
      in2 = *pRXSP0A;

      /*Used for generating test signals*/
      timeADC+=time_diff;
      freqNum = timeADC/0.5;
      adc = cos(2*PI*freq[freqNum]*timeADC);

      /*Used for reading input in real time*/
      //x1 = (double)in2;
      //adc = x1/ 2147483648.0;

      flag2=1;
}

void spdifISR(int sig)
{
      sysreg_bit_clr(sysreg_LIRPTL,SP4IMSK);
      /*Check if input is ready to avoid core hand*/
      while((*pSPCTL0>>30)!=3)
      {}
    /*Acquire input sample from SP0A register (must be done twice for
      stability)*/
      in = *pRXSP0A;
      in = *pRXSP0A;

      x2 = (double)in;
      spdif = x2/ 2147483648.0;
      /*Use this if input is needed to be buffered for analysis*/
      /*
      spdif_buf[count]=spdif;
      if (count==1024)
            count=count;*/

      sysreg_write(sysreg_LIRPTL,SP4IMSK);

}

void modulate(void)
{
      //clock_start=clock();  //used for timing
      //s = adc - h;  //used with adc input

      /*Subtract previous h from input*/
      s = spdif - h;
      /*G filter*/
      g = iir(s,a_G,b_G,u_G);
      /*Quantize output of G filter*/
      y = quantize(g,thr_Lo, thr_Hi);
      /*H filter*/
      h = iir(y,a_H,b_H,u_H);

      /*Use this for timing*/
      //clock_stop=clock();
      //secs = ((double) (clock_stop - clock_start)) / CLOCKS_PER_SEC;
    //printf("Time taken is %e seconds\n",secs);

      flag1=0;
}

/***********************************************
*
*     Timer ISR
*
***********************************************/

void timerISR(int sig)
{
      doTriState(triState);  //Update digital outputs
      /*This is used for testing*/
      /*
    if(freqNum==19)
      {
            freqNum=0;
            timeADC=0;
            //order++;
      }

      if(order>5)
            order=3;   //break here

      if(order!=orderPrev)  //if the order has changed
      {
            initVars();
            updateCoeffs();
      }
      orderPrev=order;
      */

      modulate();
}

/************************************************
*
*     Quantizer
*
************************************************/

double quantize(double inp, double low, double high)
{
      int out;
      if((inp<low)&&(!flag))
      {
            out = min;
            triState=0;
      }
      else if((inp>high)&&(!flag))
      {
            out = max;
            triState=1;
      }
      else
      {
            out = 0.0;
            triState=2;
      }
      return out;
}

void doTriState(int status)
{
      if(triState==0)
      {
            SRU(LOW,DAI_PB16_I);  //turn off LED 7
            SRU(HIGH,DAI_PB15_I);   //light LED 6
      }
      else if(triState==1)
      {
            SRU(HIGH,DAI_PB16_I);  //light LED 7
            SRU(LOW,DAI_PB15_I);  //turn off LED 6
      }
      else if(triState==2)
      {
            SRU(LOW,DAI_PB16_I);  //turn off LED 7
            SRU(LOW,DAI_PB15_I);  //turn off LED 6
      }
}


/************************************************
*
*     General IIR Filter
*
```

```
************************************************/

double iir(double iirin, const double *a, const double *b, double *u)
{
      int i;
      double out=0;
      /*Circulate buffer*/
      for(i=order; i>0; i--)
            u[i]=u[i-1];

      u[0] = iirin*a[0];

      /*Tap with a coefficients*/
      for(i=1;i<=order;i++)
            u[0]-=a[i]*u[i];
      /*Tap with b coefficients*/
      for(i=0;i<=order;i++)
            out+= b[i]*u[i];

      return out;
}
```

init1835viaSPI.c

```
////////////////////////////////////////////////////////////////////////////////////
/
//NAME:     init1835viaSPI.c (Block-based Talkthrough)
//DATE:     7/29/05
//PURPOSE:  Talkthrough framework for sending and receiving samples to the AD1835.
//
//USAGE:    This file contains the subroutines for accessing the AD1835 control
//          registers via SPI.
//
////////////////////////////////////////////////////////////////////////////////////
//
#include "tt.h"
#include "ad1835.h"

/* Setup the SPI pramaters here in a buffer first */
unsigned int Config1835Param [] = {

            WR | DACCTRL1 | DACI2S | DAC24BIT | DACFS96,
            WR | DACCTRL2 ,//| DACMUTE_R4 | DACMUTE_L4,
            WR | DACVOL_L1 | DACVOL_MAX,
            WR | DACVOL_R1 | DACVOL_MAX,
            WR | DACVOL_L2 | DACVOL_MAX,
            WR | DACVOL_R2 | DACVOL_MAX,
            WR | DACVOL_L3 | DACVOL_MAX,
            WR | DACVOL_R3 | DACVOL_MAX,
            WR | DACVOL_L4 | DACVOL_MAX,
            WR | DACVOL_R4 | DACVOL_MAX,
            WR | ADCCTRL1 | ADCFS96,
            WR | ADCCTRL2 | ADCI2S | ADC24BIT,
            WR | ADCCTRL3 | IMCLKx2

        } ;

volatile int spiFlag ;
```

```
//Set up the SPI port to access the AD1835
void SetupSPI1835 ()
{
    /* First configure the SPI Control registers */
    /* First clear a few registers     */
    *pSPICTL = (TXFLSH | RXFLSH) ;
    *pSPIFLG = 0;
    *pSPICTL = 0;

    /* Setup the baud rate to 500 KHz */
    *pSPIBAUD = 100;

    /* Setup the SPI Flag register to FLAG3 : 0xF708*/
    *pSPIFLG = 0xF708;

    /* Now setup the SPI Control register : 0x5281*/
    *pSPICTL = (SPIEN | SPIMS | MSBF | WL16 | TIMOD1) ;

}

//Disable the SPI Port
void DisableSPI1835 ()
{
    *pSPICTL = (TXFLSH | RXFLSH);
}

//Send a word to the AD1835 via SPI
void Configure1835Register (int val)
{
    *pTXSPI = val ;
    Delay(100);

    //Wait for the SPI to indicate that it has finished.
    while (1)
    {
        if (*pSPISTAT & SPIF)
            break ;
    }
    Delay (100) ;
}

//Receive a register setting from the AD1835
unsigned int Get1835Register (int val)
{
    *pTXSPI = val ;
    Delay(100);

    //Wait for the SPI port to indicate that it has finished
    while (1)
    {
        if (SPIF & *pSPISTAT)
            break ;
    }
    Delay (100) ;
    return *pRXSPI ;
//  return i ;
}

//Set up all AD1835 registers via SPI
void Init1835viaSPI()
{
    int configSize = sizeof (Config1835Param) / sizeof (int) ;
    int i ;
    SetupSPI1835 () ;

    for (i = 0; i < configSize; ++i)
    {
        Configure1835Register (Config1835Param[i]) ;
    }

    DisableSPI1835 () ;

}

//Delay loop
void Delay (int i)
{
    for (;i>0;--i)
        asm ("nop;") ;
}
```

initPLL_SDRAM.c

```
/*************************************************************************************
**
**
**  File:   initPLL.c
**  Date:   7-29-05
**  Author: SH
**  Use:    Initialize the DSP PLL for the required CCLK and HCLK rates.
**  Note:   CLKIN will be 24.576 MHz from an external oscillator.  The PLL is
**           programmed
**          to generate a core clock (CCLK) of 331.776 MHz - PLL multiplier = 27 and
**          divider = 2.
**
************************************************************************************/
#include <def21369.h>
#include <cdef21369.h>

void InitPLL_SDRAM(){

/***********************************************************************************/

int i, pmctlsetting;

//Change this value to optimize the performance for quazi-sequential accesses
//(step > //1)
#define SDMODIFY 1

    pmctlsetting= *pPMCTL;
    pmctlsetting &= ~(0xFF); //Clear

    // CLKIN= 24.576 MHz, Multiplier= 27, Divisor= 2, CCLK_SDCLK_RATIO 2.
    // Core clock = (24.576 MHz * 27) /2 = 331.776 MHz
    pmctlsetting= SDCKR2|PLLM32|PLLD2|DIVEN;
    *pPMCTL= pmctlsetting;
    pmctlsetting|= PLLBP;
    *pPMCTL= pmctlsetting;

    //Wait for around 4096 cycles for the pll to lock.
    for (i=0; i<4096; i++)
          asm("nop;");

    *pPMCTL ^= PLLBP;       //Clear Bypass Mode
```

```
*pPMCTL |= (CLKOUTEN);  //and start clkout


// Programming SDRAM control registers and enabling SDRAM read optimization
// CCLK_SDCLK_RATIO= 2.5
// RDIV = ((f SDCLK X t REF )/NRA) - (tRAS + tRP )
// (166*(10^6)*64*(10^-3)/4096) - (7+3) = 2583

*pSDRRC= (0xA17)|(SDMODIFY<<17)|SDROPT;

//=====================================================================
//
// Configure SDRAM Control Register (SDCTL) for PART MT48LC4M32B2
//
//  SDCL3  : SDRAM CAS Latency= 3 cycles
//  DSDCLK1: Disable SDRAM Clock 1
//  SDPSS  : Start SDRAM Power up Sequence
//  SDCAW8 : SDRAM Bank Column Address Width= 8 bits
//  SDRAW12: SDRAM Row Address Width= 12 bits
//  SDTRAS7: SDRAM tRAS Specification. Active Command delay = 7 cycles
//  SDTRP3 : SDRAM tRP Specification. Precharge delay = 3 cycles.
//  SDTWR2 : SDRAM tWR Specification. tWR = 2 cycles.
//  SDTRCD3: SDRAM tRCD Specification. tRCD = 3 cycles.
//
//--------------------------------------------------------------------

*pSDCTL= SDCL3|DSDCLK1|SDPSS|SDCAW8|SDRAW12|SDTRAS7|SDTRP3|SDTWR2|SDTRCD3;

// Note that MS2 & MS3 pin multiplexed with flag2 & flag3.
// MSEN bit must be enabled to access SDRAM, but LED7 cannot be driven with sdram
*pSYSCTL |=MSEN;

// Mapping Bank 2 to SDRAM
// Make sure that jumper is set appropriately so that MS2 is connected to
// chip select of 16-bit SDRAM device
*pEPCTL |=B2SD;
*pEPCTL &= ~(B0SD|B1SD|B3SD);

//=====================================================================
//
// Configure AMI Control Register (AMICTL0) Bank 0 for the ISSI IS61LV5128
//
//  WS2 : Wait States = 2 cycles
//  HC1  : Bus Hold Cycle (at end of write access)= 1 cycle.
//  AMIEN: Enable AMI
//  BW8  : External Data Bus Width= 8 bits.
//
//--------------------------------------------------------------------

//SRAM Settings
*pAMICTL0 = WS2|HC1|AMIEN|BW8;

//=====================================================================
//
// Configure AMI Control Register (AMICTL) Bank 1 for the AMD AM29LV08
//
//  WS23 : Wait States= 23 cycles
//  AMIEN: Enable AMI
//  BW8  : External Data Bus Width= 8 bits.
//
//--------------------------------------------------------------------

//Flash Settings
```

```
    *pAMICTL1 = WS23|AMIEN|BW8;

}
```

initSPORT.c

```
/////////////////////////////////////////////////////////////////////////////////////
//NAME:     initSPORT.c (Block-based Talkthrough)
//DATE:     7/29/05
//PURPOSE:  Talkthrough framework for sending and receiving samples to the AD1835.
//
//USAGE:    This file uses SPORT0 to receive data from the ADC and transmits the
//                  data to the DAC's via SPORT1A, SPORT1B, SPORT2A and SPORT2B.
//                  DMA Chaining is enabled
//
/////////////////////////////////////////////////////////////////////////////////////

#include "tt.h"

/*
   Here is the mapping between the SPORTS and the DACS
   ADC -> DSP  : SPORT0A : I2S
   DSP -> DAC1 : SPORT1A : I2S
   DSP -> DAC2 : SPORT1B : I2S
   DSP -> DAC3 : SPORT2A : I2S
   DSP -> DAC4 : SPORT2B : I2S
*/


void InitSPORT()
{
    //Clear the Mutlichannel control registers
    *pSPMCTL0 = 0;
    *pSPMCTL1 = 0;
    *pSPMCTL2 = 0;
    *pSPCTL0 = 0 ;
    *pSPCTL1 = 0 ;
    *pSPCTL2 = 0 ;

    //============================================================
    //
    // Configure SPORT 0 for input from SPDIF
    //
    //------------------------------------------------------------


    *pSPCTL0 = (OPMODE | SLEN32 | SPEN_A);


    //============================================================
    //
    // Configure SPORT2B for output to DAC 4
    //
    //------------------------------------------------------------


    *pSPCTL2 = SPTRAN | OPMODE | SLEN32 | SPEN_B;

}
```

initSRU.c

```
void InitSRU(){

   // Disable the pull-up resistors on all 20 pins
    *pDAI_PIN_PULLUP = 0x000FFFFF;

   //  Set up SPORT 0 to receive from the SPDIF receiver

   //  Tie the pin buffer input LOW.
    SRU(LOW,DAI_PB18_I);

   //  Tie the pin buffer enable input LOW
    SRU(LOW,PBEN18_I);

    //  Connect the SPDIF Receiver
    SRU(DAI_PB18_O,DIR_I);

    //  Clock in from SPDIF RX
    SRU(DIR_CLK_O,SPORT0_CLK_I);

    //  Frame sync from SPDIF RX
    SRU(DIR_FS_O,SPORT0_FS_I);

    //  Data in from SPDIF RX
    SRU(DIR_DAT_O,SPORT0_DA_I);

    // Clock on pin 7
    SRU(DIR_CLK_O,DAI_PB07_I);

    // Frame sync on pin 8
    SRU(DIR_FS_O,DAI_PB08_I);

    // Tie the pin buffer enable inputs HIGH to drive DAI pins 7 and 8
    SRU(HIGH,PBEN07_I );
    SRU(HIGH,PBEN08_I );


//------------------------------------------------------------------------------
//
//  Connect the DACs: The codec accepts a BCLK input from DAI pin 13 and
//          a LRCLK (a.k.a. frame sync) from DAI pin 14 and has four
//          serial data outputs to DAI pins 12, 11, 10 and 9
//
//          Connect DAC1 to SPORT1, using data output A
//          Connect DAC2 to SPORT1, using data output B
//          Connect DAC3 to SPORT2, using data output A
//          Connect DAC4 to SPORT2, using data output B
//
//          Connect MCLK from SPDIF to DAC on DAI Pin 6
//
//          Connect the clock and frame sync inputs to SPORT1 and SPORT2
//          should come from the SPDIF RX on DAI pins 7 and 8, respectively
//
//          Connect the SPDIF RX BCLK and LRCLK out to the DAC on DAI
//          pins 13 and 14, respectively.
//
//          All six DAC connections are always outputs from the SHARC
//          so tie the pin buffer enable inputs all high.
//

//-------------------------------------------------------------------------

//-------------------------------------------------------------------------
//  Connect the pin buffers to the SPORT data lines
```

```
    SRU(SPORT2_DB_O,DAI_PB09_I);
    SRU(SPORT2_DA_O,DAI_PB10_I);
    SRU(SPORT1_DB_O,DAI_PB11_I);
    SRU(SPORT1_DA_O,DAI_PB12_I);


    SRU(LOW,SPORT2_DB_I);
    SRU(LOW,SPORT2_DA_I);
    SRU(LOW,SPORT1_DB_I);
    SRU(LOW,SPORT1_DA_I);


//------------------------------------------------------------------------
//  Connect the clock, frame sync, and MCLK from the SPDIF RX directly
//    to the output pins driving the DACs.

    SRU(DIR_CLK_O,DAI_PB13_I);
    SRU(DIR_FS_O,DAI_PB14_I);
    SRU(DIR_TDMCLK_O,DAI_PB06_I);

//------------------------------------------------------------------------
//  Connect the SPORT clocks and frame syncs to the clock and
//    frame sync from the SPDIF receiver

    SRU(DIR_CLK_O,SPORT1_CLK_I);
    SRU(DIR_CLK_O,SPORT2_CLK_I);
    SRU(DIR_FS_O,SPORT1_FS_I);
    SRU(DIR_FS_O,SPORT2_FS_I);

//------------------------------------------------------------------------
//  Tie the pin buffer enable inputs HIGH to make DAI pins 9-14 outputs.
    SRU(HIGH,PBEN06_I);
    SRU(HIGH,PBEN09_I);
    SRU(HIGH,PBEN10_I);
    SRU(HIGH,PBEN11_I);
    SRU(HIGH,PBEN12_I);
    SRU(HIGH,PBEN13_I);
    SRU(HIGH,PBEN14_I);
//--------------------------------------------------------------------------
// Route SPI signals to AD1835.

    SRU(SPI_MOSI_O,DPI_PB01_I)      //Connect MOSI to DPI PB1.
    SRU(DPI_PB02_O, SPI_MISO_I)     //Connect DPI PB2 to MISO.
    SRU(SPI_CLK_O, DPI_PB03_I)      //Connect SPI CLK to DPI PB3.
    SRU(SPI_FLG3_O, DPI_PB04_I)     //Connect SPI FLAG3 to DPI PB4.
//---------------------------------------------------------------------------
// Tie pin buffer enable from SPI peripherals to determine whether they are
// inputs or outputs

    SRU(SPI_MOSI_PBEN_O, DPI_PBEN01_I);
    SRU(SPI_MISO_PBEN_O, DPI_PBEN02_I);
    SRU(SPI_CLK_PBEN_O, DPI_PBEN03_I);
    SRU(SPI_FLG3_PBEN_O, DPI_PBEN04_I);

//----------------------------------------------------------------------------

    *pDIRCTL=0x0;
}
```

Appendix H Raw Zero Input SNR Data

			Calculated Input Frequency [Hz]								
		Mod. Order	1021.380637	2011.180398	3006.809189	4048.443791	5055.516755	6062.600132	7130.909535	8076.741181	9083.819801
Zero Input SNR [dB]	Fs = 1MHz	1	109.6811959	108.4170688	107.9050478	109.6557744	109.7792657	107.8489634	110.2122014	108.0521061	109.8451153
		2	110.1311723	108.8479978	108.5200213	109.9312065	110.1845344	108.2209902	110.5888429	107.8665156	109.9766897
		3	110.0725683	109.09755	108.1500011	110.2297207	110.0640865	108.6649729	110.614034	107.3225259	110.2584778
		5	110.1270384	108.9845803	108.3453092	110.0920354	110.2207978	108.271864	110.6412766	107.9287235	110.0522568
		5	108.551514	109.2113906	108.1786679	110.2630905	110.1036717	108.6982807	110.6507609	107.3690793	110.2927466
	Fs = 1.5MHz	1	109.6436816	108.4088908	107.8920751	109.6551223	109.7658771	107.8675555	110.234399	108.0721001	109.8819995
		2	110.1335569	108.8658444	108.5347095	109.924806	110.1642235	108.1969863	110.5382637	107.8015119	109.8926668
		3	110.1015534	109.1141757	108.1505234	110.2368385	110.073334	108.6659973	110.6151487	107.3048251	110.2380462
		4	110.1525886	109.0008258	108.3540794	110.1166196	110.241971	108.3097763	110.6759248	107.9679341	110.1101024
		5	110.0917623	109.1097346	108.1545126	110.2495396	110.0792114	108.6872848	110.6459237	107.3657432	110.3193945
	Fs = 2 MHz	1	109.5505667	108.3597229	107.8627106	109.6301365	109.7708343	107.8328247	110.2098826	108.0549283	109.8487934
		2	110.130457	108.8384793	108.4916349	109.9082415	110.1636481	108.2027451	110.5677171	107.8521573	109.9576154
		3	110.0709215	109.0821841	108.1275152	110.2214812	110.0630902	108.6608418	110.6139757	107.322337	110.2577079
		4	110.1173099	108.9673497	108.3244406	110.0821602	110.2039593	108.2604342	110.6262016	107.9138531	110.0411029
		5	110.0856022	109.1067824	108.1416205	110.231447	110.066583	108.6630319	110.6232027	107.3221021	110.2719308

			Calculated Input Frequency [Hz]								
		Mod. Order	10090.89701	11097.97306	12202.32574	13112.1247	14119.20544	15126.2802	16133.35859	17183.57377	18147.50825
Zero Input SNR [dB]	Fs = 1MHz	1	108.7852919	108.5684128	109.7349073	106.3802404	109.2428226	108.2057232	107.6784247	108.9313795	106.1825289
		2	109.5111204	108.2771771	110.0069844	106.6198588	109.6550082	107.6196064	108.5106747	108.5547364	106.3094278
		3	109.6000112	108.3745181	110.1199565	106.7478297	109.5832149	108.5209021	107.9609186	109.0598234	105.0398678
		5	109.6096358	108.3899086	110.1478109	106.7947124	109.6426688	108.6077244	108.0658845	108.8427453	106.6243425
		5	109.2298222	109.0081121	110.1743367	106.8198671	109.6735463	108.6182088	108.1043592	109.2311933	105.2713385
	Fs = 1.5MHz	1	108.824477	108.6227728	109.7840039	106.4328469	109.3106368	108.2691141	107.7421987	109.0157113	106.2807087
		2	109.4101881	108.1487483	109.8638793	106.4416991	109.4572272	107.3831463	108.2574686	108.2755911	106.0093903
		3	109.5598867	108.3122045	110.0223072	106.6111622	109.3999943	108.2712339	107.6436203	108.6561111	104.5443889
		4	109.6771095	108.4772578	110.2417559	106.8900626	109.7477061	108.692733	108.136279	108.8647152	106.5868251
		5	109.2777558	109.0971447	110.3072078	107.0105673	109.9454137	108.9907119	108.5524801	109.762346	105.8067404
	Fs = 2 MHz	1	108.7915357	108.5794316	109.7520634	106.3940431	109.2652827	108.2273733	107.7028774	108.9563514	106.2125422
		2	109.4952218	108.2555675	109.9909702	106.605064	109.6353085	107.599669	106.4308875	108.5362034	106.2998792
		3	109.6029164	108.371615	110.1180231	106.7449161	109.5883613	108.5186501	107.9680163	109.0585921	105.0380087
		4	109.5964571	108.3783126	110.1316601	106.7707121	109.6263602	108.5839466	108.0550368	108.8148808	106.6100713
		5	109.2083875	108.9746173	110.1505187	106.7919852	109.6456585	108.6111372	108.0936657	109.2303444	105.2617975

Appendix I Raw SNR Data

			Calculated Input Frequency [Hz]							
		Mod. Order	**1021.380637**	**2011.180398**	**3006.809189**	**4048.443791**	**5055.516755**	**6062.600132**	**8076.741181**	**9083.819801**
SNR [dB]	**Fs = 1MHz**	**1**	45.11185579	45.1283063	44.57499079	44.74611476	44.48741657	43.33566513	42.03667766	39.82854259
		2	48.52110592	49.84604009	49.50754583	48.94168831	47.59261534	46.67456707	44.81198711	41.38742344
		3	48.67642241	50.48020152	50.05277782	48.84496852	47.68242669	46.68420422	44.39861298	41.42024456
		4	49.93942597	49.3985185	48.10052365	46.84680781	45.29959701	44.08988301	42.19800099	39.55067913
		5	39.67618539	20.38901237	37.36827115	36.01869444	35.02029214	34.02292038	32.72307649	32.03488525
	Fs = 1.5MHz	**1**	42.01622108	40.02929696	39.14195948	39.26291878	37.28515753	36.73752466	35.90380529	34.55846934
		2	46.6359909	45.31614001	43.91373786	42.31473878	40.93525846	39.48989248	37.40336365	35.8221983
		3	49.54748641	46.82188055	45.13279777	42.9409379	41.5046868	39.96659701	37.80528285	36.22771631
		4	45.19656196	44.33523051	43.08824367	41.7985055	40.49269606	39.29852906	37.32390467	35.72588541
		5	42.41109909	41.59930343	41.10999433	40.10143825	39.03726705	37.89730429	36.21006825	35.01824908
	Fs = 2 MHz	**1**	45.64372969	45.25181425	44.47678009	44.86772662	44.48387615	43.70013119	42.72760648	40.29678721
		2	49.9884773	49.95813394	49.97302048	49.0411312	47.96898856	47.10328637	45.67171538	41.78203038
		3	51.06688962	50.75162027	50.3210815	49.46055811	48.28433733	47.34136958	45.08828145	41.56424127
		4	48.99619662	47.40696298	45.8454424	44.3732176	42.94515873	41.67705799	39.7758403	37.77490722
		5	47.8661796	45.87257717	43.58606267	41.95979008	40.24123863	38.89526349	36.63734673	35.35195263

			Calculated Input Frequency [Hz]						
		Mod. Order	**10090.89701**	**11097.97306**	**13112.1247**	**14119.20544**	**15126.2802**	**16133.35859**	**18147.50825**
SNR [dB]	**Fs = 1MHz**	**1**	40.66938466	40.11916614	37.72239743	37.29532094	36.66082765	34.22067851	32.84548213
		2	42.2812354	42.02694197	38.55368715	38.08962389	37.30955833	34.91913909	33.17488866
		3	42.22739531	41.60872717	38.52103184	37.68199409	37.11743204	34.77137347	33.02247022
		4	39.95134157	39.3145724	36.46741191	36.06176293	35.63339136	33.46482288	31.63662214
		5	35.30038062	34.61102677	32.67494166	32.14343882	31.56534833	31.16371781	29.8145695
	Fs = 1.5MHz	**1**	35.10853828	34.0891824	32.81302988	32.08145258	31.72036182	30.72799625	29.73643597
		2	36.49919737	35.51318765	33.4911325	32.69768548	31.88882609	31.22739526	30.03764716
		3	36.76988843	35.88391395	33.79473596	32.89700974	32.18071099	31.31167996	29.93089446
		4	36.60633641	35.82349632	33.80657574	33.18880957	32.32690477	31.67940938	30.00983463
		5	35.88712666	35.20260688	33.40034301	32.87123674	32.19609142	31.82077719	30.53577605
	Fs = 2 MHz	**1**	40.49892246	39.94296068	36.07862889	35.6631873	35.02538852	33.98347548	32.38952531
		2	42.31302908	41.69394297	36.91929319	36.06245578	35.14009325	6.365561313	32.93939297
		3	41.89209581	41.34109078	36.46673816	35.85621443	35.09703022	34.14171796	32.43354893
		4	38.41364068	37.89530949	34.36401473	33.71370442	32.67250404	31.91225108	30.46235922
		5	36.97095605	36.08317943	33.60201221	32.89738264	32.20051836	31.38515246	30.14309011

Appendix J Raw THD Data

			Calculated Input Frequency [Hz]							
		Mod. Order	1021.380637	2011.180398	3006.809189	4048.443791	5055.516755	6062.600132	8076.741181	9083.819801
THD [%]	Fs = 1MHz	1	0.595839564	0.210036637	0.167681573	0.25510573	0.330239785	0.254963466	0.198988689	0.155521664
		2	0.642315698	0.273972212	0.300281601	0.255184086	0.273160215	0.225934107	0.124619787	0.09142683
		3	0.576708512	0.261997858	0.094688025	0.194661044	0.247684062	0.190985383	0.11587523	0.120361126
		5	0.513236589	0.258545981	0.20707789	0.421656348	0.454820321	0.297869693	0.333565415	0.289440518
		5	0.611272762	0.7819743	0.291146511	2.221477614	2.235278736	2.344135475	2.063564628	1.941519167
	Fs = 1.5MHz	1	0.592220948	0.294553535	0.315758524	0.21520284	0.675011374	0.394869601	0.13174512	0.168508828
		2	0.605277116	0.274229744	0.153296515	0.214278641	0.25211169	0.159991036	0.133065656	0.161652004
		3	0.606690224	0.253659329	0.233309626	0.169469166	0.27335408	0.167850853	0.210610278	0.193810264
		4	0.609920048	0.258200601	0.292791603	0.145158168	0.301547474	0.281387241	0.208925207	0.253158481
		5	0.637072965	0.239267247	0.238479613	0.298650742	0.419411385	0.316351905	0.418865267	0.297551428
	Fs = 2 MHz	1	0.551950237	0.278757244	0.249300928	0.257268057	0.443992342	0.080091682	0.250565478	0.121954907
		2	0.646676239	0.271400445	0.283237132	0.231930366	0.340603269	0.246760626	0.068372137	0.153521964
		3	0.588155243	0.221936155	0.295009838	0.288549141	0.103363168	0.166266046	0.091777081	0.169056606
		4	0.660338914	0.265421734	0.237723196	0.664798118	0.517642692	0.610229266	0.664266902	0.598589308
		5	0.728995341	0.215158924	0.201649579	0.795031303	0.930970083	1.121075334	1.210670802	1.090125027

			Calculated Input Frequency [Hz]						
		Mod. Order	10090.89701	11097.97306	13112.1247	14119.20544	15126.2802	16133.35859	18147.50825
THD [%]	Fs = 1MHz	1	0.144252938	0.114893785	0.096079322	0.177254248	0.150393157	0.107295422	0.171374893
		2	0.074107754	0.125552106	0.098478809	0.113615559	0.128547883	0.09128277	0.151202445
		3	0.131682253	0.090100616	0.068203648	0.094873495	0.083374717	0.106448278	0.143572194
		5	0.291480844	0.236487136	0.156439981	0.149975352	0.126378335	0.182020218	0.205196706
		5	1.596007168	1.366384252	0.687822199	0.419073763	0.122119255	0.124232019	0.330840696
	Fs = 1.5MHz	1	0.126368359	0.191749032	0.2155077	0.235604872	0.097832233	0.158438504	0.130392319
		2	0.103702829	0.189606084	0.134036555	0.192298529	0.130371744	0.113015111	0.154499559
		3	0.159089053	0.197214554	0.172642818	0.193562418	0.213520373	0.199969486	0.178334979
		4	0.228513686	0.266649252	0.14680173	0.271029539	0.236868867	0.135253512	0.277786764
		5	0.201450061	0.399279449	0.248959677	0.236346661	0.245537648	0.149689537	0.247485814
	Fs = 2 MHz	1	0.0987823	0.160845388	0.128247895	0.128681362	0.042391419	0.112743292	0.131550531
		2	0.135956516	0.092565682	0.1227844	0.170120242	0.083570373	0.270310642	0.202953122
		3	0.11220544	0.073428429	0.091723296	0.096713236	0.121612743	0.122592347	0.170781935
		4	0.568751276	0.429900091	0.199252527	0.212066967	0.199579787	0.11626091	0.166366855
		5	0.972642312	0.800839034	0.574013041	0.345858419	0.155208077	0.16206899	0.319165873

Appendix K Raw Power and Efficiency Data

Input Freq. [Hz]	Mod. Freq. [MHz]	Mod. Order	Input Amplitude	Vin [Volts]	I in [Amps]	Pin [watts]	Rout [Ohms]	Vout (RMS) [V]	Pout [watts]	Efficiency	Efficiency [%]	Average Efficiency
20	2	1	2	39.3	2.16	84.888	8.07	25.85	82.8033	0.975441567	97.54416	
100	2	1	2	39.7	2.19	86.943	8.07	26.08	84.2833	0.969408934	96.94089	
800	2	1	2	39.7	2.2	87.34	8.07	26.15	84.7364	0.97018971	97.01897	
1500	2	1	2	39.7	2.18	86.546	8.07	26.04	84.025	0.970870767	97.08708	
4000	2	1	2	39.7	2.19	86.943	8.07	26.2	85.0607	0.978350399	97.83504	
12000	2	1	2	39.7	1.97	78.209	8.07	24.85	76.5208	0.978413685	97.84137	
20000	2	1	2	39.8	1.45	57.71	8.07	21	54.6468	0.946921507	94.69215	0.969942367
20	1.5	1	2	39.1	2.34	91.494	8.07	27.05	90.6695	0.990987986	99.0988	
100	1.5	1	2	39.6	2.4	95.04	8.07	27.44	93.3028	0.981721386	98.17214	
800	1.5	1	2	39.6	2.39	94.644	8.07	27.55	94.0524	0.993748726	99.37487	
1500	1.5	1	2	39.6	2.4	95.04	8.07	26.93	89.8668	0.945567952	94.5568	
4000	1.5	1	2	39.6	2.4	95.04	8.07	26.74	88.6032	0.932272435	93.22724	
12000	1.5	1	2	39.6	2.22	87.912	8.07	26.26	85.4508	0.97200332	97.20033	
20000	1.5	1	2	39.7	1.79	71.063	8.07	22.92	65.0962	0.916035183	91.60352	0.961762427
20	1	1	2	39.3	2.15	84.495	8.07	25.74	82.1001	0.971656007	97.1656	
100	1	1	2	39.4	2.17	85.498	8.07	25.76	82.2277	0.961750071	96.17501	
800	1	1	2	39.4	2.17	85.498	8.07	25.92	83.2523	0.97373438	97.37344	
1500	1	1	2	39.4	2.17	85.498	8.07	25.81	82.5472	0.965487196	96.54872	
4000	1	1	2	39.4	2.17	85.498	8.07	25.93	83.3166	0.974485863	97.44859	
12000	1	1	2	39.5	1.94	76.63	8.07	24.44	74.0166	0.965895278	96.58953	
20000	1	1	2	39.6	1.43	56.628	8.07	20.79	53.5594	0.945810695	94.58107	0.965545641
20	2	2	2	38.1	2.31	88.011	8.07	26.19	84.9958	0.965740638	96.57406	
100	2	2	2	39.5	2.4	94.8	8.07	27.32	92.4885	0.975617357	97.56174	
800	2	2	2	39.7	2.41	95.677	8.07	27.4	93.031	0.97234423	97.23442	
1500	2	2	2	39.7	2.41	95.677	8.07	27.48	93.575	0.978030442	97.80304	
4000	2	2	2	39.8	2.41	95.918	8.07	27.32	92.4885	0.964245766	96.42458	
12000	2	2	2	39.8	2.12	84.376	8.07	25.65	81.527	0.966233901	96.62339	
20000	2	2	2	39.8	1.49	59.302	8.07	21.17	55.5352	0.93648072	93.64807	0.965527579
20	1.5	2	2	38.1	2.31	88.011	8.07	26.26	85.4508	0.970909953	97.091	
100	1.5	2	2	39.6	2.41	95.436	8.07	27.41	93.0989	0.975511308	97.55113	
800	1.5	2	2	39.9	2.44	97.356	8.07	27.63	94.5994	0.971685033	97.1685	
1500	1.5	2	2	39.9	2.44	97.356	8.07	27.65	94.7364	0.973092252	97.30923	
4000	1.5	2	2	39.9	2.44	97.356	8.07	27.55	94.0524	0.966066338	96.60663	
12000	1.5	2	2	40	2.1	84	8.07	25.35	79.631	0.947988582	94.79886	
20000	1.5	2	2	39.9	1.47	58.653	8.07	20.93	54.2831	0.92549631	92.54963	0.961535682
20	1	2	2	38	2.29	87.02	8.07	26.29	85.6461	0.984211779	98.42118	
100	1	2	2	39.6	2.41	95.436	8.07	27.32	92.4885	0.969115694	96.91157	
800	1	2	2	39.6	2.41	95.436	8.07	27.35	92.6918	0.971245228	97.12452	
1500	1	2	2	39.7	2.42	96.074	8.07	27.44	93.3028	0.971155573	97.11556	
4000	1	2	2	39.8	2.39	95.122	8.07	27.51	93.7794	0.98588594	98.58859	
12000	1	2	2	39.8	2.12	84.376	8.07	25.61	81.2729	0.963222656	96.32227	
20000	1	2	2	39.9	1.49	59.451	8.07	21.16	55.4827	0.933251352	93.32514	0.968298318

Input Freq. [Hz]	Mod. Freq. [MHz]	Mod. Order	Input Amplitude	Vin [Volts]	Iin [Amps]	Pin [watts]	Rout [Ohms]	Vout (RMS)	Pout [watts]	Efficiency	Efficiency [%]	Average Efficiency
20	2	5	1.5	39.8	1.38	54.924	8.07	20.55	52.3299	0.952769748	95.27697	
100	2	5	1.5	39.8	1.39	55.322	8.07	20.63	52.7382	0.953294416	95.32944	
800	2	5	1.5	39.9	1.38	55.062	8.07	20.59	52.5338	0.954085238	95.40852	
1500	2	5	1.5	39.8	1.38	54.924	8.07	20.53	52.2281	0.950916111	95.09161	
4000	2	5	1.5	39.9	1.37	54.663	8.07	20.45	51.8219	0.948024644	94.80246	
12000	2	5	1.5	39.9	1.24	49.476	8.07	19.45	46.8776	0.947482278	94.74823	
20000	2	5	1.5	39.9	0.95	37.905	8.07	16.93	35.5173	0.937009255	93.70093	0.949083099
20	2	4	1.5	39.8	1.37	54.526	8.07	20.5	52.0756	0.955059762	95.50598	
100	2	4	1.5	39.8	1.36	54.128	8.07	20.47	51.9233	0.95926847	95.92685	
800	2	4	1.5	39.8	1.37	54.526	8.07	20.42	51.6699	0.947620182	94.76202	
1500	2	4	1.5	39.8	1.37	54.526	8.07	20.49	52.0248	0.954128224	95.41282	
4000	2	4	1.5	39.8	1.37	54.526	8.07	20.53	52.2281	0.957857105	95.78571	
12000	2	4	1.5	39.8	1.24	49.352	8.07	19.13	45.3478	0.91886487	91.88649	
20000	2	4	1.5	39.8	0.94	37.412	8.07	16.33	33.0445	0.88325867	88.32587	0.939436755
20	2	3	1.5	39.8	1.38	54.924	8.07	20.57	52.4318	0.95462519	95.46252	
100	2	3	1.5	39.8	1.37	54.526	8.07	20.7	53.0967	0.973785979	97.3786	
800	2	3	1.5	39.8	1.37	54.526	8.07	20.54	52.279	0.958790461	95.87905	
1500	2	3	1.5	39.8	1.39	55.322	8.07	20.59	52.5338	0.949601269	94.96013	
4000	2	3	1.5	39.8	1.39	55.322	8.07	20.53	52.2281	0.944074988	94.4075	
12000	2	3	1.5	39.8	1.24	49.352	8.07	19.41	46.685	0.945960014	94.596	
20000	2	3	1.5	39.8	0.91	36.218	8.07	16.73	34.6831	0.957621488	95.76215	0.95492277
20	2	2	1.5	39.8	1.37	54.526	8.07	20.56	52.3809	0.960658537	96.06585	
100	2	2	1.5	39.8	1.38	54.924	8.07	20.54	52.279	0.951842704	95.18427	
800	2	2	1.5	39.8	1.39	55.322	8.07	20.5	52.0756	0.941317895	94.13179	
1500	2	2	1.5	39.8	1.37	54.526	8.07	20.52	52.1772	0.956924203	95.69242	
4000	2	2	1.5	39.8	1.37	54.526	8.07	20.43	51.7206	0.948548539	94.85485	
12000	2	2	1.5	39.8	1.2	47.76	8.07	19.15	45.4427	0.951480087	95.14801	
20000	2	2	1.5	39.8	0.87	34.626	8.07	16.25	32.7215	0.944997961	94.4998	0.950824275
20	2	1	1.5	39.8	1.25	49.75	8.07	19.56	47.4094	0.952952121	95.29521	
100	2	1	1.5	39.8	1.25	49.75	8.07	19.53	47.2641	0.950031197	95.00312	
800	2	1	1.5	39.8	1.25	49.75	8.07	19.57	47.4579	0.953926759	95.39268	
1500	2	1	1.5	39.8	1.27	50.546	8.07	19.54	47.3125	0.936027894	93.60279	
4000	2	1	1.5	39.8	1.26	50.148	8.07	19.47	46.9741	0.936709125	93.67091	
12000	2	1	1.5	39.8	1.13	44.974	8.07	18.39	41.9073	0.931812234	93.18122	
20000	2	1	1.5	39.8	0.84	33.432	8.07	15.46	29.6173	0.885896705	88.58967	0.935336576