EGB240 : ELECTRONIC DESIGN

ASSIGNMENT 2

LIAM PERCY

N9959807

31/5/2019

Executive Summary

The following report documents the design process of a digital voice recorder. The digital voice recorder is comprised of input conditioning circuitry, built on a breadboard, which has been mounted to a Teensy development board. All input conditioning circuitry is powered by our Teensy's 5V USB power supply.

This circuitry amplifies an electret condenser microphone signal to approximately 5Vpp, biased at 2.5V. To prevent any alias distortion from interfering with the signal, a fourth order Chebyshev filter has been cascaded with the amplification/bias circuit. A Chebyshev filter was chosen as it is able to produce the same frequency response as other filter designs at a lower order of filter. By designing a lower order filter, more accurate results were produced, based on our ideal design.

Components for this filter were tuned and selected based on E series standard component values, and then mapped to our filter using a unity-gain Sallen-Key lowpass topology. This topology provided ample degrees of freedom, such that any error associated with untuned component selection was minimised.

The filter removes all frequencies above the Nyquist limit of 7.8125kHz, determined by our ADC sample rate. This filter is also designed such that it preserves most frequency content within the human speech frequency bandwidth, between 5Hz and 2.8kHz.

An ATMega34U4 microcontroller sources the resultant voltages output by this conditioning circuitry, through an analogue to digital conversion process. This process is handled by the microcontroller's aforementioned ADC peripheral, which samples voltages at a rate of 15.625kHz.

Each digital sample is stored in a 2 page, 512 byte ring buffer (1024 bytes total). Once a given page is full of samples, the entire page is written to an SD card using the FAT filesystem, and the ADC begins filling the second page of the buffer.

Output of the recorded waveform occurs via the microcontroller's pulse width modulation (PWM) peripheral. Samples are read from the SD card into the buffer, and modulated through a comparison with our TOP value. This value represents the maximum counter value for our digital carrier waveform, and is defined at 512, such that our PWM peripheral will output at a rate of 31.25kHz. This frequency is far outside the human audible frequency range, and ensures we will not hear any high frequency noise as a result of lower PWM output frequencies. To account for this faster PWM output frequency each sample is played from the buffer twice. In doing so, each unique sample is output at the same rate in which they were recorded, and this ensures the accurate reproduction of our input signal for listening.

All DVR operations (play, record, stop) are handled using a case-based state machine defined by enumerated type values. The embedded code will continuously loop within one of these given states, unless a certain input value is determined. Playback occurs at the press of pushbutton S1, and ends after the stop button S3 is pressed, or each sample has been produced. Recording occurs at the press of pushbutton S2, and ends after 305 pages of samples (10 seconds) have been taken, or the stop button is pressed.

Contents

Tables and Figures

1 Introduction

In the field of digital recording, a microphone is used to detect continuous audio signals, which are then converted into a stream of discrete numbers representing a change in air pressure over time. These numbers can be stored using digital memory, and then retrieved and converted back to their original analogue waveforms for listening.

Figure 1: Audio Levels Display on a Digital Audio Recorder

The following report will outline this procedure in greater detail, describing the fundamental electrical concepts required at each stage of these recording, conversion and playback processes. It will also define the core methodology by which these concepts were applied to the design and development of our own digital voice recorder.

To detect and store the analogue signal, the final digital voice recorder design will implement input conditioning circuitry, interfaced to a microcontroller via the analogue to digital converter (ADC) peripheral. The recorder will then make use of the microcontrollers pulse width modulation (PWM) peripheral to output this stored signal to a simple headphone interface. Record, playback and stop commands are controlled via pushbuttons connected to the microcontroller, and during these states, a corresponding LED will be on to provide a visual representation of each operation to the user.

Noting these design specifications and characteristics, this report will serve to not only define and describe the measurable design goals of the recorder, but also outline any limitations or boundaries in place, that may be outside the scope of the project.

All aspects related to the technical design process, including operation principles, component selection and simulation analysis will be clearly documented. These aspects will the be further refined, and their implementation into the final design will be clearly outlined using assembly overlays, prototype photos and test procedures. The design, simulation and implementation of the recorder will be tested and validated through experimental results.

After employing these various systematic techniques to document and produce our final digital voice recorder design, this report will reflect upon the entire production process, identifying both successful elements and unsuccessful elements within the project. In doing so, we will be able to recognise areas within the design which may benefit from additional, improved development, or how the design could be enhanced outside of any time, technical and financial constraints.

2 Literature Review

To more clearly understand many of the technical design decisions made throughout this project, several key pieces of literature have been presented, which serve as the basis for which each decision was made. Reviewing these pieces of literature, allows us to better recognise how the execution of the project was informed.

2.1 Bandwidth of Human Speech

When designing a digital voice recorder, careful consideration must be given to the frequency range of human speech. This band of frequencies will significantly dictate how we choose to design our input conditioning circuit, and the rate in which we sample these frequencies.

The spectrum of speech covers a wide portion of the complete audible frequency spectrum. At a normal vocal intensity, the energy of vowels usually diminishes rapidly above approximately 1kHz [1]. In general, the majority of energy encapsulated in human vocal content is concentrated between 300Hz and 3kHz [2].

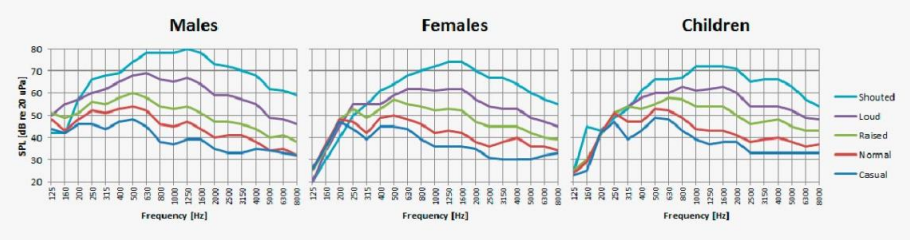


Figure 2: Voice Spectra (1/3 Octave) Depending on Efforts

When recording and playing back sound content, we are most concerned with the intelligibility of speech. Intelligibility is a measure of how comprehensible speech is for any given conditions. For non-tonal (Western) languages, the most important frequency range regarding perceived intelligibility lies around 2kHz ± 1kHz [1].

We can measure the effect that filter application has on speech intelligibility by applying both high and low-pass filters to our vocal content. In figure 3 below, we can observe that when applying a low pass filter at 2.5kHz, speech intelligibility drops by approximately 20%.

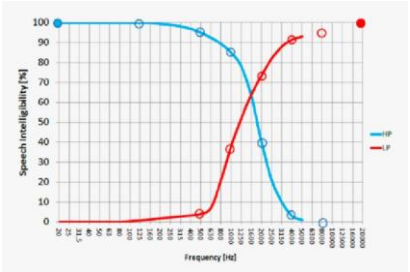


Figure 3: Speech Intelligibility When Applying LP and HP Filters

To ensure our microcontroller captures important frequency content, without compromising speech intelligibility, our input signal conditioning should be designed around these factors.

2.2 Nyquist–Shannon Sampling Theorem

The designed digital voice recorder will make use of an ADC module to convert a continuous-time (analogue) signal into a stream of discrete points, or samples, such that it may store these samples in digital memory. A higher sampling rate will produce more accurate sample data, and thus a more accurate signal reproduction, however this also produces a large volume of data, and requires very fast analogue to digital converters [3].

Considering these factors, it is important to define the minimum necessary sampling frequency for a given signal, to prevent against the distortion of underlying signal information, and allow for an accurate signal reconstruction. This definition is provided by the Nyquist-Shannon sampling theorem, which states that if the highest frequency component for a given analogue signal is given by f_{max}, then the sampling rate f_s, must be greater than $2f_{max}$ [3], [4]. Moreover, if the sampling frequency is given by f_s, then the critical frequency (or Nyquist Limit) f_n is defined $f_s/2$ [3].

Any sinusoidal component of a given signal higher than f_n will be reintroduced to the sampled signal by folding at the frequency f_n. This effect is known as aliasing and will have to be accounted for and removed using input conditioning circuitry, to ensure the reproduced signal is clear and free from higher frequency distortions.

2.3 Aliasing

As discussed, aliasing is the effect produced in which frequencies higher than the Nyquist Limit are incorrectly sampled by our ADC, and re-introduced to our sampled signal, around this limit. This effect is clearly demonstrated in figures 4 and 5 below.

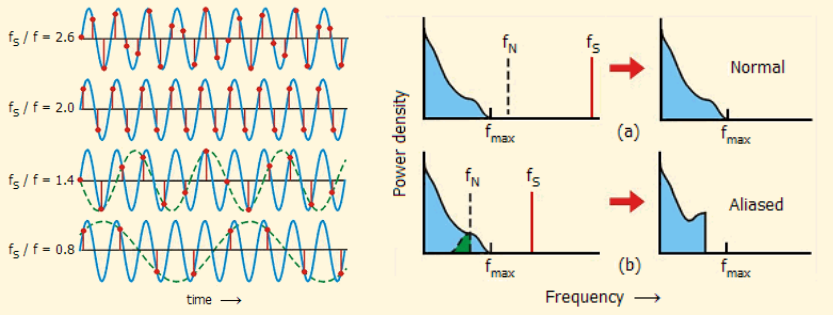


Figure 4: Sampling of Signal at Different Sampling Rates fs

Figure 5: Frequency Spectrum of Normal and Aliased Signal

Figure 4 displays a sinusoidal signal of frequency f in the time domain sampled at four different sampling frequencies f_s. In the first two cases, the sampling rates can produce an accurate reconstruction of the original signal. On the other hand, the last two cases present a situation in which the sampled points may be considered as belonging to signals of lower frequencies [3]. Figure 5 displays the frequency spectrum of a signal at two different sampling rates. In (a), the sampling frequency is sufficiently higher than f_{max}, and thus no aliasing occurs. In (b), the sampling frequency is insufficient, and thus aliasing takes place before the signal is stored in a distorted form.

2.4 Designing Higher Order Filters

To prevent the effect of aliasing occurring when sampling signals detected by our microphone, we can apply a low-pass filter to our input conditioning, such that we completely remove all frequencies above the Nyquist Limit. The problem with such a specification is that the filter may also attenuate frequencies that lie within the previously discussed bandwidth of human speech.

Filter frequency attenuation occurs within the stop band of a given filter. First order filters attenuate frequencies within this band at a rate of 20dB/decade, while second order filters attenuate at 40dB/decade [5]. As the order of the filter is increased, the stop band response of the filter approaches its ideal stop band characteristic [6].

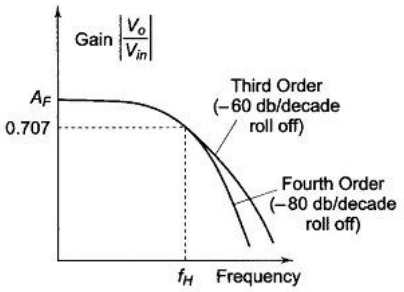


Figure 6: Third and Fourth Order Low Pass Filter Frequency Response

By designing higher order filters, we can prevent the attenuation of frequencies in the human speech band, while removing frequencies above the Nyquist limit. Higher order filters are formed by simply cascading or connecting first and second order filters in series. A fourth order filter, for example, is composed of two cascaded second order sections [6].

These higher order filters can be designed by first generating a transfer function, based on several specified filter characteristics, including the passband and stopband edge frequencies, in addition to the maximum passband ripple and minimum stopband attenuation. Given the general transfer function form:

$$H(s) = \frac{\omega^2}{s^2 + \frac{\omega_n}{Q}s + {\omega_n}^2}$$

We can factor the higher order polynomial into pole pairs, such that:

$$\frac{\omega_n}{Q} = -p_1 - p_2, \quad {\omega_n}^2 = p_1 * p_2$$

For each filter order, there will be an associated pole pair, and as such, we can use these equations to calculate both ω_n and Q for any given number of second order filter stages, before cascading them together to create higher order filters.

We are then able to map component values to our cascaded filters, using the Sallen-Key topology. The Sallen-Key topology is an electronic filter topology used to implement second order active filters [7]. It has a high input, low output impedance [8], and its transfer function is given by:

$$\frac{v_{out}}{v_{in}} = \frac{\frac{K}{R_1R_2C_1C_2}}{s^2 + s\left(\frac{1}{R_1C_2} + \frac{1}{R_2C_2} + \frac{1-K}{R_2C_1}\right) + \frac{1}{R_1R_2C_1C_2}}$$

Such that

$$\omega_n = \frac{1}{\sqrt{R_1R_2C_1C_2}}, \quad Q = \frac{\sqrt{R_1R_2C_1C_2}}{R_2C_1 + R_1C_1 + R_1C_2(1-K)}, \quad K = 1 + \frac{R_4}{R_3}$$

When applied in filter designs with high-gain accuracy and relatively low quality factors (Q), we can utilise a unity-gain Sallen-Key low pass filter, which can aid in simplifying component calculations, while still providing the degrees of freedom necessary to reduce the error associated with untuned component selection

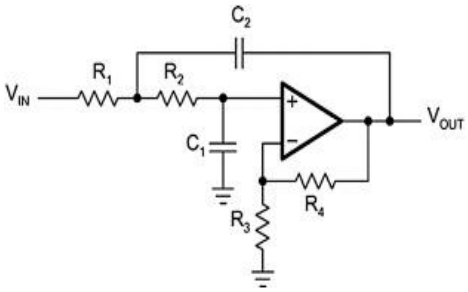


Figure 7: General Sallen-Key Low-Pass Filter

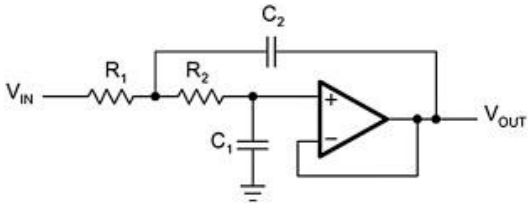


Figure 8: Unity-Gain Sallen-Key Low-Pass Filter

2.5 E-Series Standard Component Values

While designing higher order filters improves the effect of the stop band response, the accuracy of the filter will decline. More specifically, the difference between the actual stop band response and the ideal stop band response increases as we increase in the order of the filter [5].

To minimise this decreased accuracy, we can utilise E series standard component values. The E series is a system of preferred numbers, consisting of E1, E3, E6, E12, E24, E48, E96 and E192 series [9], where each number after E designates the quantity of values per decade of components. The spacing of the components in each series is based on an even logarithmic ratio defined as $Ratio = 10^{\frac{1}{N}}$ where N is the series number [10].

E1	100											
E3	100				220				470			
E6	100		150		220		330		470		680	
E12	100	120	150	180	220	270	330	390	470	560	680	820
E24	100	110	120	130	150	160	180	200	220	240	270	300
	330	360	390	430	470	510	560	620	680	750	820	910
E96	100	102	121	124	147	150	178	182	215	221	261	267
	316	324	383	392	464	475	562	576	681	698	825	845
	105	107	127	130	154	158	187	191	226	232	274	280
	332	340	402	412	487	499	590	604	715	732	866	887
	110	113	133	137	162	165	196	200	237	243	287	294
	348	357	422	432	511	523	619	634	750	768	909	931
	115	118	140	143	169	174	205	210	249	255	301	309
	365	374	442	453	536	549	649	665	787	806	953	976

Figure 9: IEC-63 Nominal Resistance/Capacitance

As we can observe in figure 9 above, each E series contains a given number of steps per decade based on it's proceeding number, and adhered to the ratio $10^{\frac{1}{N}}$

E1 Series Resistance: 1Ω, 10Ω, 100Ω, 1000Ω etc.

$$E6 = 10^{\frac{1}{6}} = 1.46$$

$$\frac{330}{220} = \frac{220}{150} = \frac{150}{100} \cong 1.46$$

2.6 Pulse Width Modulation Output

Given an analogue signal that has been sampled and stored in digital memory using the microcontroller ADC, we now require a method to allow for the reconstruction and playback of this signal in its original, analogue form. For this, we will make use of the microcontrollers pulse width modulation (PWM) module.

In PWM, a digital control is used to create a square wave, or an on/off pattern. This on-off pattern can simulate voltages between full on and off by changing the portion of the time the signal spends on versus the time that the signal spends off [11]. The duty cycle of the PWM waveform describes the proportion of on time with the period time. We can modulate this duty cycle over time to produce varying analogue values.

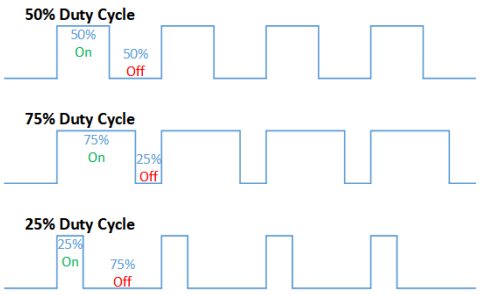


Figure 10: Various Square Wave Duty Cycles

By varying our PWM output based on our analogue signal input, we can utilise a sawtooth carrier waveform, set at the PWM frequency. By comparing the intersections of our carrier waveform and our analogue signal, we can synthesise appropriate output pulses [12], [13].

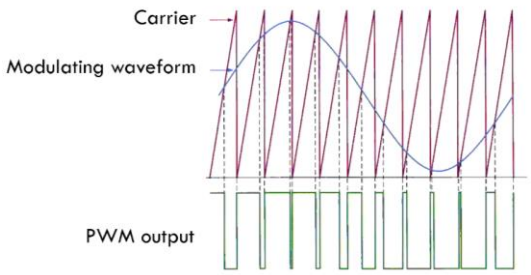


Figure 11: Carier Waveform Used to Synthesise PWM Output

It must be noted that the above representation compares carrier and input signals in the continuous domain, however this process will occur on our microcontroller in the digital domain. Our analogue signal has already been converted into a digital representation using our ADC at this point in the design. To produce a digital representation of our carrier waveform, we can utilise a counter within our microcontroller. The counter will increment with each time step and reset after a given period. We can synthesise our PWM output much in the same way as we did in the analogue domain, by comparing a given digital sample with our counter value, and outputting a pulse based on this value.

This process is represented to the right in figure 12, were the counter is represented by each step with period T. The PWM period is equal to the TOP value (maximum counter value) multiplied by each counter period, and the OCR value represents a digital sample the we can compare with our counter [12].

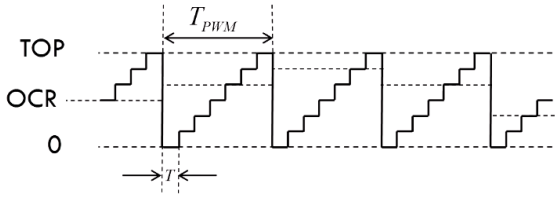


Figure 11: Digital Carrier Waveform

If we compare every sample stored in our digital memory, we can accurately reproduce our recorded signal in its original analogue form, using this modulated output PWM signal.

3 Project Description

To summarise, this project encompasses the design and production of a simple digital voice recorder (DVR). The overall system can be separated or divided into various stages, each of which performs a separate, individual function. The final digital voice recorder is the result of interfacing these various stages with one-and-other.

The recorder begins by detecting and amplifying an analogue signal, through a simple electret condenser microphone connected to one of four operational amplifiers encapsulated in an LMC6484 IC package.

This signal is then sent through a low pass filter, acting to remove any frequencies above our given Nyquist Limit, as discussed in section 2.2 of this report. In doing so, we ensure that we prevent aliasing distortion from being introduced into our sampled signal, as discussed in section 2.3 of this report.

The Nyquist rate is determined by the ADC sampling frequency, which is set by our microcontroller timer. This project makes use of the QUT-designed development board, which features an Atmel ATMega34U4 microcontroller mounted to a Teensy microcontroller board, SD memory card for storing digital samples, pushbuttons for user input and LEDs as output indicators (see Appendix 1).

The development board has been fitted alongside a breadboard, in which the analogue input conditioning circuit has been built upon. This circuitry, consisting of our microphone, filter and amplifiers, will rely on the 5V power supply provided by the Teensy's USB power connector. The circuitry will produce an output signal that will be sent to the Teensy boards JIN pin, where it will be converted into digital samples using the ADC peripheral.

The operation of the digital voice recorder, namely the record and playback functions, are serviced through firmware written to the microcontroller, in the form of imbedded C code. This code checks for user input on three push buttons, S1, S2, and S3, each of which is connected to an associated PIN on the microcontroller. S1 is located on PINF4 and begins the playback of any recorded audio stored on our SD card. The green LED1, located on PIND4 will be on, and serve as a visual output indicator during playback. Similarly, S2 is located and PINF5 and handles recording functionality, while S3 is located on PINF6, and is responsible for stopping both recording and playback functionality. LED2 on PIND5 and LED3 on PIND6 serve as visual indicators for while recording and stopped respectively.

Finally, during the aforementioned playback functionality, the microcontroller will make use of a simple headphone interface kit, connected to the Teensy JOUT pin, in conjunction with the microcontrollers pulse width modulation (PWM) peripheral, to reproduce, or output a stored signal, back into its analogue form. The process in which this conversion takes place is outlined in section 2.6 of this report.

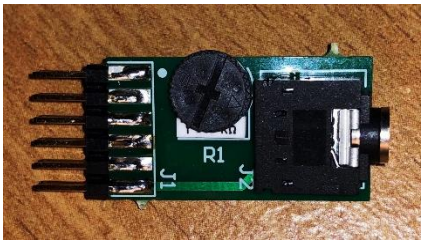


Figure 12: Headphone Interface Kit

After the successful implementation of each of these design stages, they can be interfaced with each other, to produce our final digital voice recorder design. A block diagram in figure 12 is displayed below, to provide a visual representation of the final system.

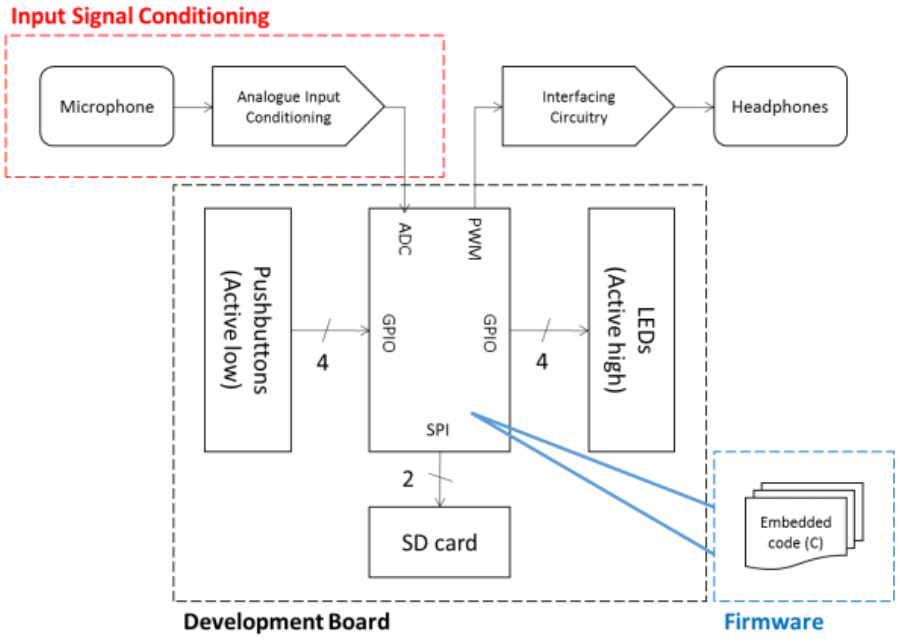


Figure 13: Digital Voice Recorder Block Diagram

4 Design Goals & Specification

There are a number of important design goals and measurable specifications encapsulated within this project. By outlining these goals and specifications, we are able to better understand the overall design problem. We can also assess which goals should be classified as a mandatory requirement for the overall success of the design, and which elements may fall outside of the project scope.

4.1 Input Conditioning Circuitry

The input conditioning circuitry elements within the DVR design are responsible for several important aspects, vital to the success of the design. We can define these under three categories; amplification, anti-aliasing and biasing.

4.1.1 Biasing

There are two primary goals involving biasing with our input condition circuitry. At a minimum, the circuitry will have to both remove any DC bias output via the microphone and bias the signal itself at half the ADC supply voltage, or 2.5V.

We can bias the microphone output using a voltage divider. If we use equal resistance values in our divider, our 5V supply will be split into 2.5V, and we can now use this voltage as a reference to 'ground' in our amplification stage. In doing so, the amplifier will output a signal around 2.5V. We will need to consider the impedance on both amplification terminals v_n and v_p, and ensure they are the same, such that they cancel each other out, and we reduce any errors or losses within our system.

DC bias can be removed by connecting a capacitor in series with our microphone. Due to the way our microphone and amplifier must be connected, we will always see some attenuation of lower frequencies, as the component configuration creates a high-pass filter. Based on factors of human speech frequency bandwidth discussed in section 2.1 of this report, this capacitor value will have to carefully considered, and designed in such a way that it does not attenuate speech content within this human speech frequency spectrum.

4.1.2 Amplification

We can deduce the output voltage characteristic of our microphone by analysing its specifications listed within the datasheet (Appendix 2). Our microphone is rated with a sensitivity of -42 ± 2dB, which equates to approximately 10mV. Based on this reading, to produce an output closer to 5V, we will require a gain factor approximately 500.

Resistance values utilised to produce our amplifier will need to be chosen such that they minimise the current flowing into out Op-Amp terminals. This serves to reduce error, and as discussed in the previous section, 4.1.1, we can match the current flow into each terminal, such that any error produced is cancelled out.

4.1.3 Anti-Aliasing

The ADC peripheral in our microcontroller is configured to sample signals at 15.625kHz. As discussed in section 2.2 of this report, any frequencies above the Nyquist Limit of our sampling frequency, which in this case is given by $f_s/2$, or 7.8125kHz, will result in the introduction of aliasing frequencies to out recorded signal.

To remove these frequencies, we need to apply a low pass filter that completely attenuates all frequencies after this point. In addition, based on the characteristics of human speech frequency content outlined in section 2.1 of this report, we must ensure that our filter does not attenuate frequencies within this frequency range.

Ideally, our filter would keep *all* frequencies that fall within the bandwidth of human speech, however this goal will likely fall outside of this projects scope, based on both technical and design limitations that occur when implementing the increasingly higher order magnitude filters required to meet this characteristic.

For this reason, the decision was made to take a more deliberated and conservative approach when implementing our anti-aliasing filter design, with the intension of preserving only the most important frequency content associated with human speech.

4.2 Embedded Code

The embedded code stored on our microcontroller is responsible for handling the overall operation of our digital voice recorder. There are several key aspects and specifications that define how this code should behave, based on various inputs provided to it by the user.

The most evident requirement of our embedded code is its response to pushbuttons. Each pushbutton, it's associated PIN, and it's corresponding LED visual output indicator were outlined in the project description (section 3) of this report, however the actions of these buttons can be elaborated on and specified further.

4.2.1 Playback

Pushbutton S1 is responsible for handling the playback of any recorded audio stored on our memory card. If the play button is held on beyond the end of the recording, playback will cease, and a new playback cycle will not be initiated until the play button is released and then pressed again. In addition, record functionality will be disabled during playback operation.

4.2.2 Recording

Pushbutton S2 is responsible for initiating the recording functionality of our DVR. Recording will continue until either the stop button is pressed, or the maximum record time is reached. Similar to our playback button, if the record button is held beyond the maximum time limit, recording will cease, and a new record cycle will not be initiated until the record button is released. Playback functionality should be disabled while recording.

4.2.3 Stop

Pushbutton S3 is responsible for immediately ceasing any playback or recording cycle initiated through pushbuttons S1 and S2. This will also turn off any LEDs associated with either playback or recording, such that only the blue LED3 will be active.

4.3 PWM Output

Once playback is initialised, the digital signal stored on our SD memory card will be output to a simple headphone interface, using the Teensy's PWM peripheral. The process for PWM output was outlined in detail throughout section 2.6 of this report. To ensure the success of our DVR design, there are two main specifications for PWM output that must be considered.

Firstly, our PWM output must occur at the same rate as our recorded signal, otherwise the resultant signal will be subject to distortion, and the speech intelligibility (discussed in section 2.1) will fall significantly. Secondly, we must ensure that our PWM frequency does not fall within the audible range of human hearing (approximately 20Hz to 20kHz). If the frequency *does* fall within this range, it will be interfering with the reproduction of our analogue signal by producing a single tone noise, at its defined frequency.

5 Scope

Just as we defined key goals and specifications involved within the production of our DVR, it is also important to describe the scope of the project, and therefore define limitations or boundaries for various design elements.

As alluded to in section 4.1.2 of this report, an ideal digital voice recorder would make use of a significantly higher order filter, such that a greater bandwidth of frequencies may be captured. We discussed the implications of higher order filter design/implementation in section 2.4 of this report, specifically regarding how, with an increase in the order of a filter we will observe a correlated decrease the accuracy of the filter.

In addition, every second increase of a filter order (i.e. 2^{nd}, 4^{th} etc.) requires an additional operational amplifier. While in theory we could employ an unlimited number of amplifiers to meet this specification, the technical limitations of this project do not allow for such a design choice. Because of this technical limitation, in addition to the aforementioned accuracy decrease, the decision was made to bound the order of the filter applied to a maximum of 6.

Beyond out input conditioning circuitry, the current specifications for our firmware only allow a user to record and play back a single audio file, with a maximum length of 10 seconds. Through additions to the embedded C code, it is possible to both increase this maximum file length and allow a user to record and playback multiple files. We could specify a maximum number of files to be stored on the SD card, and define pushbutton S4 such that it may delete a given recording. Perhaps double clicking the play button (pushbutton S1) could cycle through recordings.

While this additional functionality may enhance the user experience slightly, providing visual feedback to the user based on these new actions will prove to the incredibly complex, and outside of the project scope based on our technical and resource limitations.

Finally, the specifications for PWM output were discussed in section 4.3 of this report. While it is true that our PWM signal must output frequencies at the same rate as our recorded signal, this does not necessarily mean the PWM frequency has to match the sample rate of out recorded signal. Through a process known as interpolation, we can construct new data points within a range of known, discrete data points [14]. By increasing our PWM rate, we can output new samples in between our originals to better model our input signal.

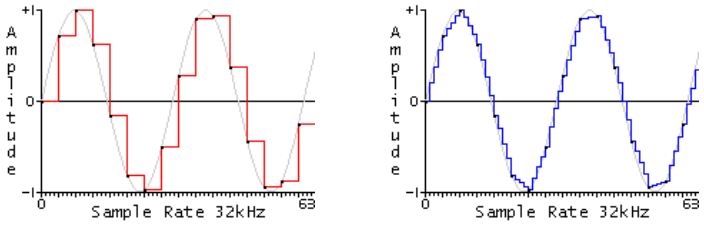


Figure 14: (Left) No Interpolation (Right) Interpolation

While this technique can improve the quality of a recorded signal, the ADC sample rate configured at 15.625kHz was deemed to be of ample quality for the purpose of this DVR design, and thus interpolation was defined as outside of the project scope.

6 Methodology

6.1 Design Process/Analysis Techniques

In order to ensure the overall success of our digital voice recorder design, it is important to define a clear procedure or methodology describing how the project will be conducted. As discussed throughout this report, our DVR is comprised of several various stages, or modules, each of which has its own clearly defined function and specifications (see section 4).

For this reason, our design process will focus on designing, prototyping, building and testing each module in stages. In doing so, we can confirm the intended behaviour and functionality of each module independently, before interfacing each module and testing the overall system.

Drawing from our design goals and specifications outlined in section 4.1, we will begin by building and testing our input conditioning circuitry. This process will involve component selection based on the E Series standard component values discussed in section 2.5 of this report, in addition to both simulation and experimental testing, to analyse and confirm our theoretical calculations and component selections. Component selection will be based on the Sallen-Key topology discussed in section 2.4 of this report. Filter transfer functions and calculations will take place within a MATLAB environment.

Once we have validated the operation of our input conditioning circuit, we will begin designing and developing our embedded C code, which handles the operation and output of our digital voice recorder. We will begin by assigning LEDs to their associated pushbutton and ensuring each push button is mapped to its correct functionality through the use of a state machine. We can test simple button handling and PWM output by using our firmware to append the PWM duty cycle, based on pushbutton presses. The result of this test can be analysed experimentally through the use of an oscilloscope, such that we may confirm the firmware is behaving as intended.

6.2 Validation Test Procedure

As discussed, the core functionality and behaviour of each module will be tested and validated in isolation, before we eventually interface these modules together and test the system as a whole. To run these isolation tests, we can make use existing firmware/circuitry provided as part of this project, that allow us to test the general functionality of each module independently.

Once we have produced our analogue conditioning circuitry, we can load a piece of testing firmware onto our microcontroller, which will allow us to record and playback audio. This firmware will provide a general confirmation that our circuitry is designed as intended, however it does not meet all our necessary specifications outlined in section 4 of our report, and thus we will need to develop our own firmware version for the final DVR design.

Similarly, once we have developed our custom firmware, we will be able to test this firmware in isolation, with an input conditioning circuit test jig. We can load our firmware onto the Teensy within this jig, and ensure it operates as intended.

Only after validating the intended behaviour of each module in isolation, will we then interface these stages together. At this point, further testing and analysis will be performed on the overall system. Much in the same way as we have done for individual stages, we will make use of an oscilloscope and signal generator to confirm the behaviour of the overall system and ensure the success of the final design.

7 Technical Design

7.1 Input Conditioning Circuitry

7.1.1 Biasing

While the microphone sensitivity was previously deduced to be 10mV, this was in relation to a 0dB reference of 1V. In practice, our microphone sensitivity was closer to 50mV pk-pk, and thus we required a gain factor of 100, to amplify this signal up to 5V.

The technical aspects of our amplification process will be discussed in the next section of this report, however for the purposes of biasing, we will be using an operational amplifier configured as an inverting op-amp. The negative terminal of this amplifier will be connected to a 10kΩ resistor.

As discussed in section 4.1.1 of this report, we can use a voltage divider to bias our signal around 2.5V, by selecting resistors of equal value, and connecting this voltage divider to our amplifier's positive terminal. To ensure the impedance on both terminals v_n and v_p is equal, two 20kΩ resistors were selected to create our voltage divider. This serves to minimise any errors or losses within our system.

We also discussed how, when removing DC bias from our microphone through the use of a capacitor, the connection between the capacitor and the amplifier will create a high pass filter. The cut-off frequency of this filter is defined by

$$\omega_c = \frac{1}{R_1 C}$$

We know our R1 value is 10kΩ, and as such, we can define our cut-off frequency in rad/s, and then rearrange this equation to select a value for our capacitor. A cut-off frequency f_c of 5Hz was chosen, as this frequency will not interfere with our human speech frequency bandwidth, and also produces a suitable capacitance value for our design.

$$\omega_c = 2 * \pi * 5Hz = \frac{1}{R_1 C}$$

$$C = \frac{1}{R_1 \omega_c} = \frac{1}{10E3 * 2 * \pi * 5} = 3.2\mu F$$

7.1.2 Amplification

As discussed previously, our microphone output characteristic sits at approximately 50mV. To amplify this signal up to 5V, we require a gain factor of 100. We configured our amplifier as an inverting op-amp, and choose large resistance values to minimise current flowing through the system. Noting this, we can select our components based on the gain function for inverting amplifiers:

$$G = \frac{R_2}{R_1}, \quad 100 = \frac{R_2}{R_1} = \frac{1M\Omega}{10k\Omega}$$

With out amplification and bias stage of circuitry completed, we simulated our circuit behaviour, and analysed the results.

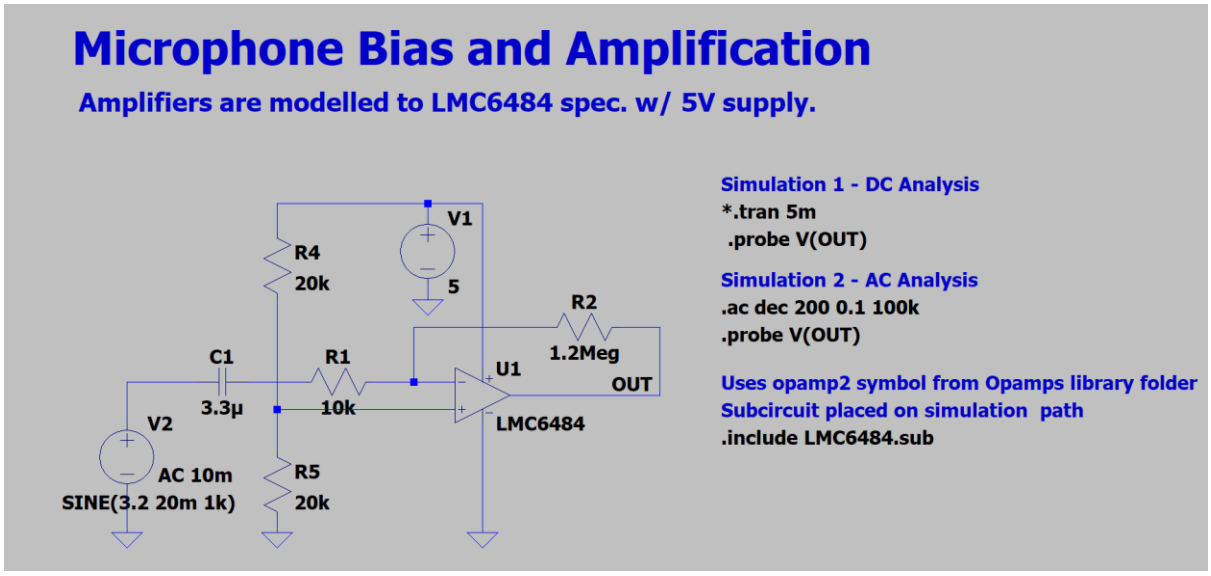


Figure 15: Microphone Bias and Amplification Circuit

The above circuit models the behaviour of our amplification and bias conditioning circuitry. The netlist for this circuit will be displayed later in this report, when simulating the input conditioning circuitry as a whole. Simulation 1 returns the following:

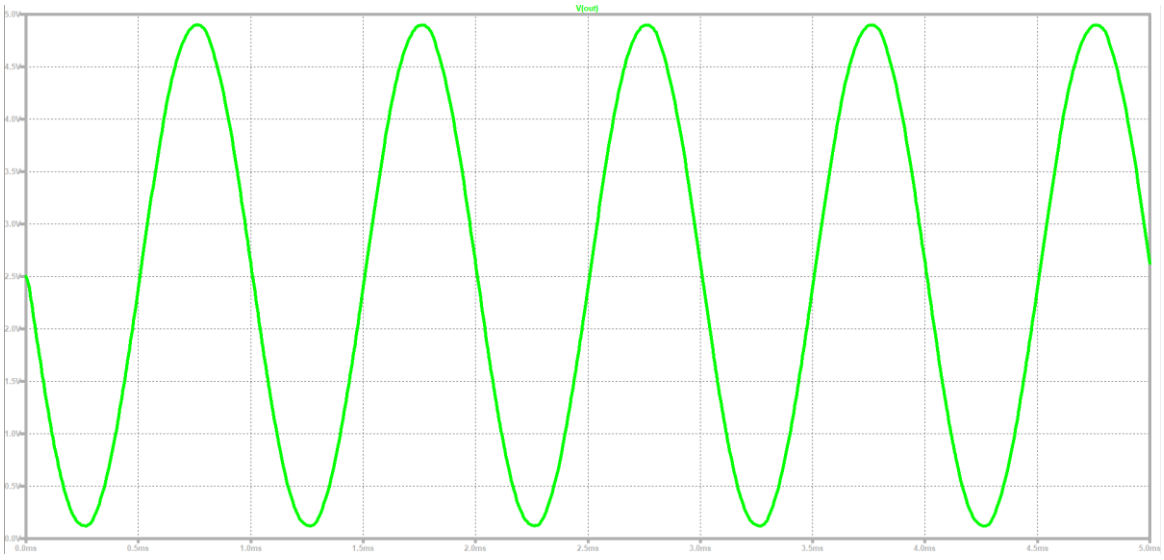


Figure 16: Microphone Bias and Amplification DC Analysis

Simulation 2 returns the following:

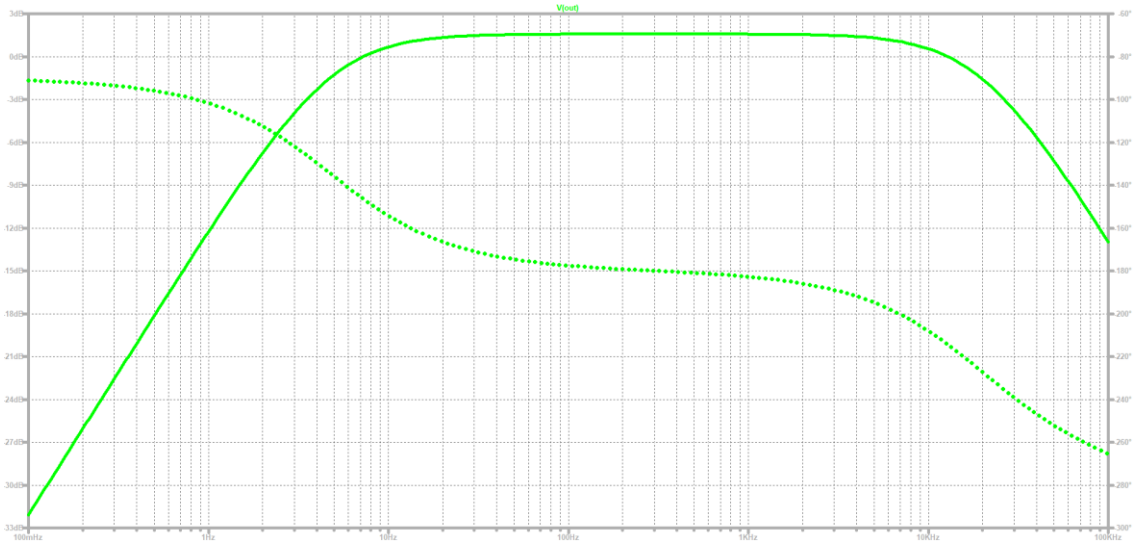


Figure 17: Microphone Bias and Amplification AC Analysis

As we can observe in figure 16, our signal is amplified to approximately 5V pk-pk, around 2.5V. Any further amplification to this signal would result in clipping. Essentially, our microphone signal would be amplified above the 5V limit, which would in a distorted sample being sent to our microcontroller.

Figure 17 displays the AC analysis of our circuit. We can observe that our high pass filter begins to roll off frequencies around the 10Hz mark and has a -3dB point at 5Hz. This meets the specifications we designed for previously, ensuring that our amplifier and bias circuitry does not attenuate any frequencies within the human speech frequency bandwidth. Further, we can observe that around these frequencies of interested, a fairly low amount phase distortion is introduced, however this will not significantly affect the final signal captured by our ADC. Finally, we can observe a roll-off at around 20kHz. This occurs because our operation amplifier does not have an unlimited band-width, and thus our gain will decrease at higher frequencies. This effect is known as the gain-bandwidth product, however as it is affecting frequencies outside of our capture range, it can be ignored for this system.

Our gain level at our frequencies of interest is slightly above 0dB. The 0dB point represents a 40dB gain from our microphones -40dB input, or a gain factor of 100 (*20*log(40)*). Our gain is slightly above this point, to account for the slight attenuation present in our Chebyshev filter passband, due to it's ripple characteristic.

7.1.3 Anti-Aliasing

7.1.3.1 Design

As discussed in section 4.1.3 of this report, the ADC peripheral in our microcontroller is configured to sample signals at 15.625kHz. To prevent any aliasing signals from being introduced into our sample, we will need to filter out all frequencies above the Nyquist Limit of this sample frequency, which sits at 7.8125kHz.

To remove these frequencies from our input signal, without attenuating frequencies within the human speech frequency bandwidth, we employed the various techniques discussed in section 2.4 of the report, pertaining to the design of higher order filters.

To begin, MATLAB was used to design and model an ideal filter, based on a number of defined conditions.

```
fsamp = 15.625e3;           % Sample rate, Hz
fs = fsamp/2;               % Stopband frequency, Hz
fp = 2780;                  % Passband frequency, Hz

wp = 2*pi*fp;               % Stopband frequency, rad/s
ws = 2*pi*fs;               % Passband frequency, rad/s

Amin = 20*log10(1/2^8);     % min. stopband attentuation, dB
Amax = 1.3;                 % max. passband attentuation, dB
```

A Chebyshev filter design was chosen because, after testing various designs, it was found that this filter type is able to meet the same specifications defined by these values when compared to a Butterworth filter, with two less filter orders. Lower order filters produce more accurate results based on their ideal model, and as such, we can specify key design characteristics with much more confidence. Our passband value of 2780Hz was chosen with this factor in mind, and also that of the human speech frequency bandwidth. By choosing this value, in accordance research outlined in section 2.1 of this report, this frequency will maintain a speech intelligibility of 90%+, enough to meet our required specifications

Using these defined values, we could then use MATLABs inbuild *cheb1ord* function, which returns the order N of the lowest order analogue Chebyshev Type I filter which has a passband ripple of no more than Amax dB and a stopband attenuation of at least Amin dB.

```
[n_cheb, wn_cheb] = cheb1ord(wp, ws, Amax, Amin, 's');
```

This command returns the following:

```
n_cheb  =     4

wn_cheb =     1.7467e+04
```

Indicating that with these specifications, we will produce a fourth order Chebyshev filter, with a natural frequency of 17 467 rad/s. We can pass the values to MATLABs *cheby1* function, which designs our low pass filter. It will return two resultant vectors, which we can then use to model a transfer function for a filter, and also plot it's frequency response.

```
[b, a] = cheby1(n_cheb, Amax, wn_cheb, 'low', 's');

H = tf(b, a);       % Define transfer function
bode(H);            % Plot transfer function freq resp.
```

This code produces the following results:

```
H =

                          1.97e16

  -------------------------------------------------------

  s^4 + 1.506e04 s^3 + 4.186e08 s^2 + 3.471e12 s + 2.288e16
```

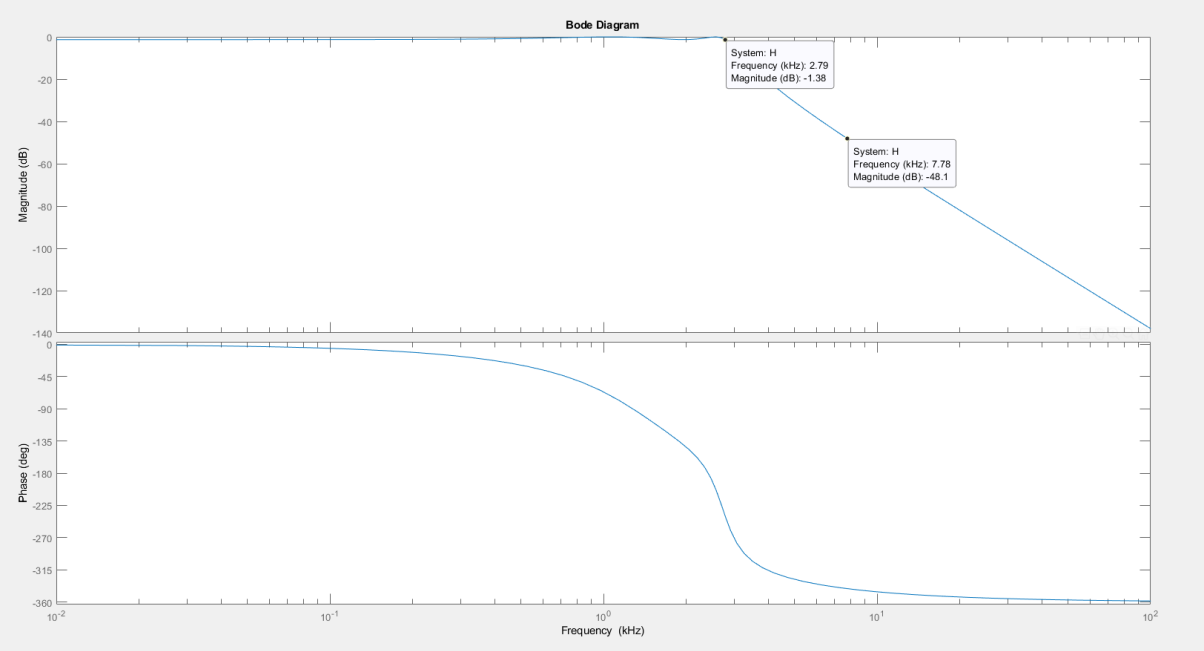


Figure 18: Bode Plot of Filter Transfer Function

As we can observe from the figure above, our filter produces a gain of approximately -1.3dB at 2.8kHz, and -48dB at 7.8125kHz, which match exactly to our pre-defined specifications. Further, we can analyse the passband frequency response more closely, to deduce the effects of the Chebyshev ripple.

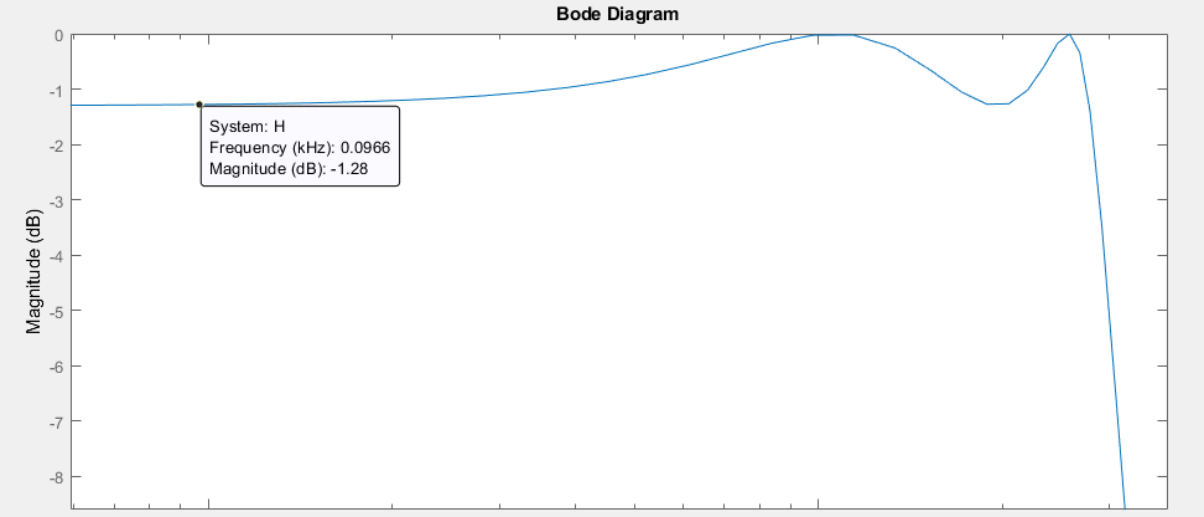


Figure 19: Chebyshev Ripple Characteristic Analysis

In the figure above we can observe that the system attenuates all frequencies in the passband by approximately 1.28dB, except towards the edge of the passband, where the filter exerts it's ripple characteristic. This attenuation has been accounted for in the amplification stage of our circuit, as described in section 7.1.2. In doing so, some frequencies now experience a voltage gain, however this effect has been accounted for. We can observe in figure 16 that the DC amplification of our microphone is slightly below 5V. This provides enough room for these frequency gains to occur, without causing any clipping distortion in our final DVR design.

With our filter characteristics designed, and any effects of attenuation in the passband accounted for, we can now utilise our Sallen-Key transfer function, in order define a relationship between possible component values, and further, map these values to our Sallen-Key topology. Given the Sallen-Key transfer function:

$$\frac{v_{out}}{v_{in}} = \frac{\frac{K}{R_1 R_2 C_1 C_2}}{s^2 + s\left(\frac{1}{R_1 C_2} + \frac{1}{R_2 C_2} + \frac{1-K}{R_2 C_1}\right) + \frac{1}{R_1 R_2 C_1 C_2}}$$

We can define a relationship between our resistors and capacitors, and also model the function based on unity gain, such that

$$C_2 = nC_1, \quad R_2 = mR_1$$

and

$$\omega_n = \frac{1}{RC\sqrt{mn}}, \quad Q = \frac{\sqrt{mn}}{(1+m)}$$

We can also define a relationship between n and m

$$n = \frac{Q^2(1+m)^2}{m}$$

We can calculate these two values for each 2nd order filter stage by factoring our transfer MATLAB transfer function to find our pole pairs. This process is outlined in greater detail throughout section 2.4 of this report. We can then cascade these 2nd order filter stages together, in order to produce our 4th order Chebyshev filter.

```
[z, p, k] = tf2zpk(b, a);   % Factor polynomial to find pole pairs

c1 = -p(1)-p(2);            % Calculate wn/Q (1)
c2 = p(1)*p(2);             % Calculate wn^2 (1)
c3 = -p(3)-p(4);            % Calculate wn/Q (2)
c4 = p(3)*p(4);             % Calculate wn^2 (2)

wn1 = sqrt(c2)              % Calculate wn (1)
Q1 = wn1/c1                 % Calculate Q (1)

wn2 = sqrt(c4)              % Calculate wn (2)
Q2 = wn2/c3                 % Calculate Q (2)
```

This code produced the following outputs.

```
wn1 =

   1.7136e+04

Q1 =

    3.8842

wn2 =

   8.8266e+03

Q2 =

    0.8287
```

7.1.3.2 Component Selection

The selection of components for use in our filter is based on the variables found in the previous section of the report, in addition to E Series standard component values discussed in section 2.5. The quality of our recorded signal will very closely depend on the accuracy of this anti-aliasing filter, and as such it is important that we are very conservative with our component selection, and that we always account for tolerancing issues that may arise between each component.

An excel spreadsheet was produced (appendix 3), allowing us to tune our component selection in a very tight, conservative manner. An m ratio was produced by listing all potential E24 series resistors, and by substituting this value into the equation,

$$n = \frac{Q^2(1+m)^2}{m}$$

,alongside our defined Q value. Using this formula, we then found an n ratio between capacitors. With these two ratios defined, we were very easily able to select our appropriate R and C values, to meet the required design specifications. This process was repeated for both of our second order filter stages, using our two ω_n and Q values defined above.

The final values for our Chebyshev filter were as follows:

$$R_1 = 8.2k\Omega \qquad C_1 = 5.6nF$$
$$R_2 = 16k\Omega \qquad C_2 = 18nF$$
$$R_3 = 5.1k\Omega \qquad C_3 = 1nF$$
$$R_4 = 10k\Omega \qquad C_4 = 68nF$$

7.1.3.3 Simulation

Using these values, a simulation circuit was modelled for our filter, and the behaviour of this simulation was analysed to ensure it met the projects required specifications.

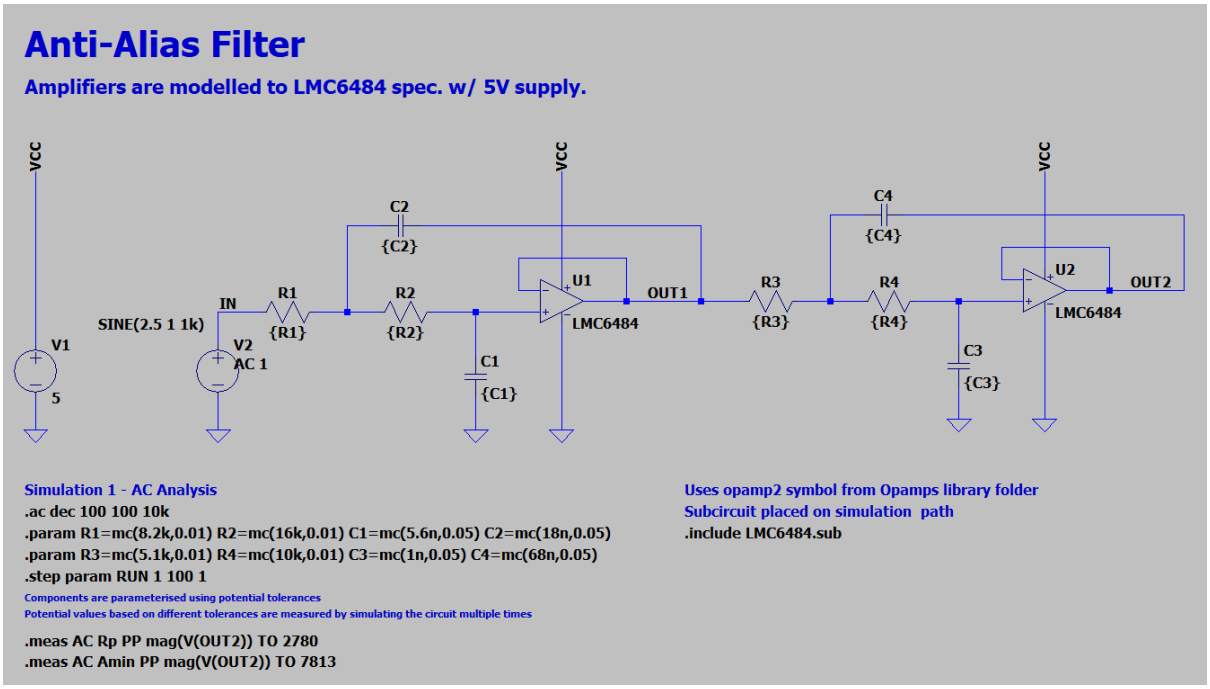


Figure 20: Anti-Aliasing Filter Circuit

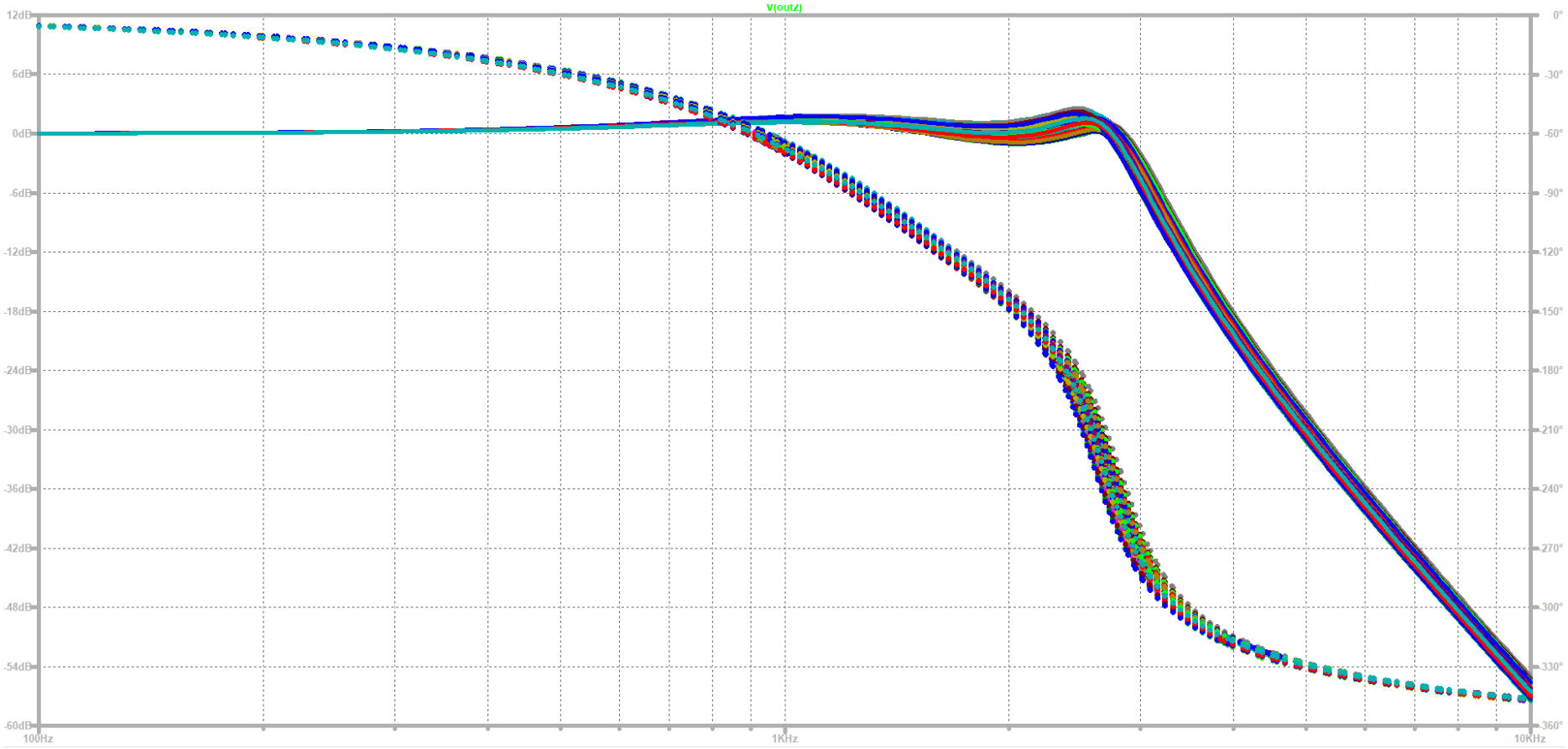


Figure 21: Anti-Aliasing Filter Frequency Response

As we can observe from the figure above, our practical fourth order Chebyshev filter behaves very similar to our theoretical model, at least in these simulations. As part of our simulations, we modelled each component as a possible set of values, based on component tolerances. With these possible values, we than run our simulation multiple times, such that we are able to ensure our filter meets the required specifications, even under a worst-case scenario. We used two measurement commands without our simulation, to store the gain amount at our two critical passband and stopband locations.

These measurements appear for each simulation, in the below form:

```
Measurement: rp
  step      PP(mag(v(out2)))   FROM   TO
     1      (2.36948dB,0°)     100    2780
     2      (1.51336dB,0°)     100    2780
     3      (2.72215dB,0°)     100    2780
     4      (2.09375dB,0°)     100    2780
     5      (1.86808dB,0°)     100    2780

Measurement: amin
  step      PP(mag(v(out2)))   FROM   TO
     1      (49.6149dB,0°)     100    7813
     2      (48.7833dB,0°)     100    7813
     3      (49.6206dB,0°)     100    7813
     4      (48.8332dB,0°)     100    7813
     5      (49.1556dB,0°)     100    7813
```

As we can observe from these measurements above, our passband gain is not too high, nor is our stopband gain too low. That being said, these are the first 5 values of 100. As such, a more comprehensive way to ensure the design meets our required specification, is to plot each value and create a histogram based on the results.

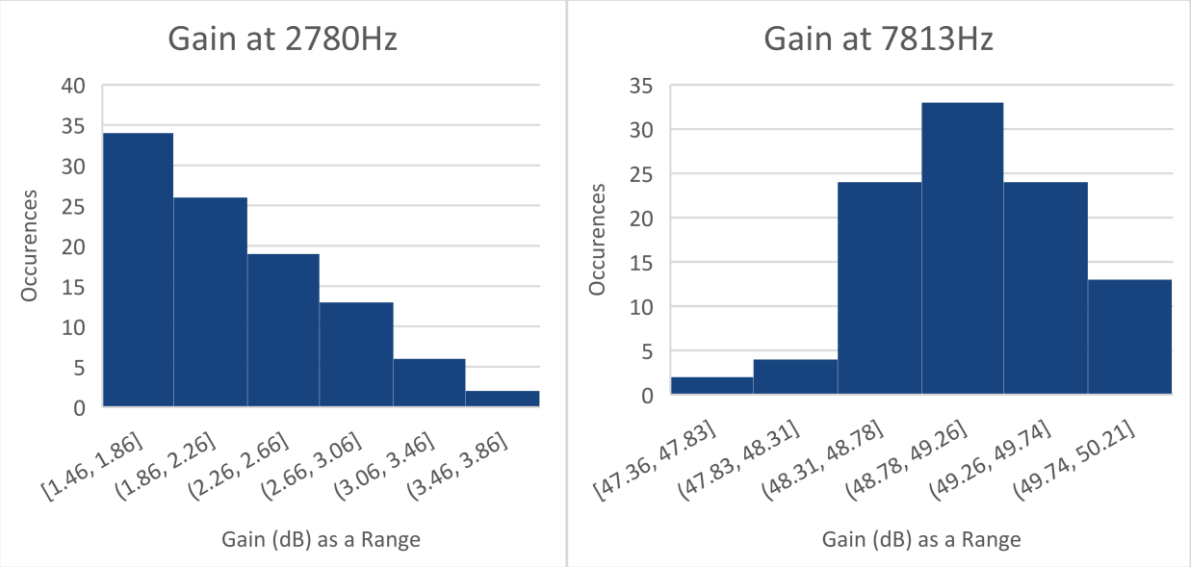


Figure 22: -ve Gain at Passband and Stopband Edges

In both cases, over 95% of the simulated gain values meet our required design specifications at our passband and stopband edges. If it is deduced that our selected component values do not meet these specifications throughout our experimental testing, they will be revisited and redesigned to better match our design intent.

Furthermore, two our filters have been connected based on their quality factor. The filter with the lower quality factor will appear first in the cascade, such that the resonance in the higher quality filter will be rolled off. This prevents clipping distortion from affecting our signal.

With our biasing, amplification and anti-aliasing circuitry complete, we can now connect these sub-circuits together to produce our complete analogue input conditioning circuitry. The schematic and simulations for this completed circuitry can be viewed below:

7.1.4 Schematic

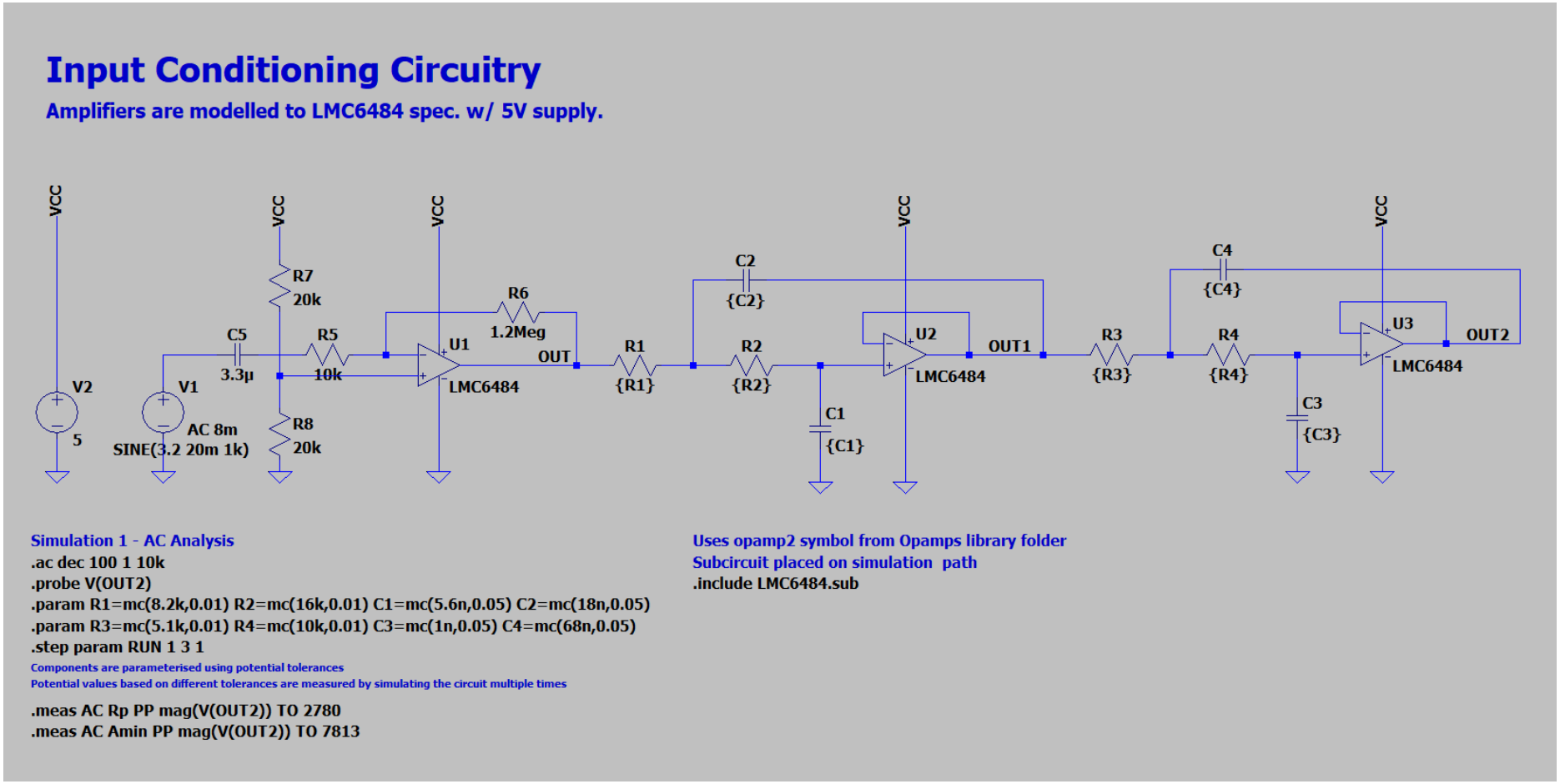


Figure 23: Input Conditioning Circuitry Simulation Schematic

7.1.5 Netlist

```
* Input Conditioning Circuitry

* Simulation 1 - AC Analysis
.ac dec 100 1 10k
.probe V(OUT2)
.param R1=mc(8.2k,0.01) R2=mc(16k,0.01) C1=mc(5.6n,0.05) C2=mc(18n,0.05)
.param R3=mc(5.1k,0.01) R4=mc(10k,0.01) C3=mc(1n,0.05) C4=mc(68n,0.05)
* Components are parameterised using potential tolerances
* Potential values based on different tolerances are measured by simulating the circuit multiple times
.meas AC Rp PP mag(V(OUT2)) TO 2780
.meas AC Amin PP mag(V(OUT2)) TO 7813
.step param RUN 1 3 1

* Amplifiers are modelled to LMC6484 spec. w/ 5V supply.
V1 N004 0 SINE(3.2 25m 1k) AC 10m
V2 VCC 0 5
XU1 N008 N003 VCC 0 OUT LMC6484
XU2 N007 OUT1 VCC 0 OUT1 LMC6484
XU3 N006 OUT2 VCC 0 OUT2 LMC6484

R1 N002 OUT {R1}
R2 N007 N002 {R2}
R3 N001 OUT1 {R3}
R4 N006 N001 {R4}
R5 N003 N005 10k
R6 OUT N003 1.2Meg
R7 VCC N008 20k
R8 N008 0 20k
C1 N007 0 {C1}
C2 OUT1 N002 {C2}
C3 N006 0 {C3}
C4 OUT2 N001 {C4}
C5 N005 N004 3.3µ

* Uses opamp2 symbol from Opamps library folder
* Subcircuit placed on simulation  path
.include LMC6484.sub

.backanno
.end
```

7.1.6 Simulation 1

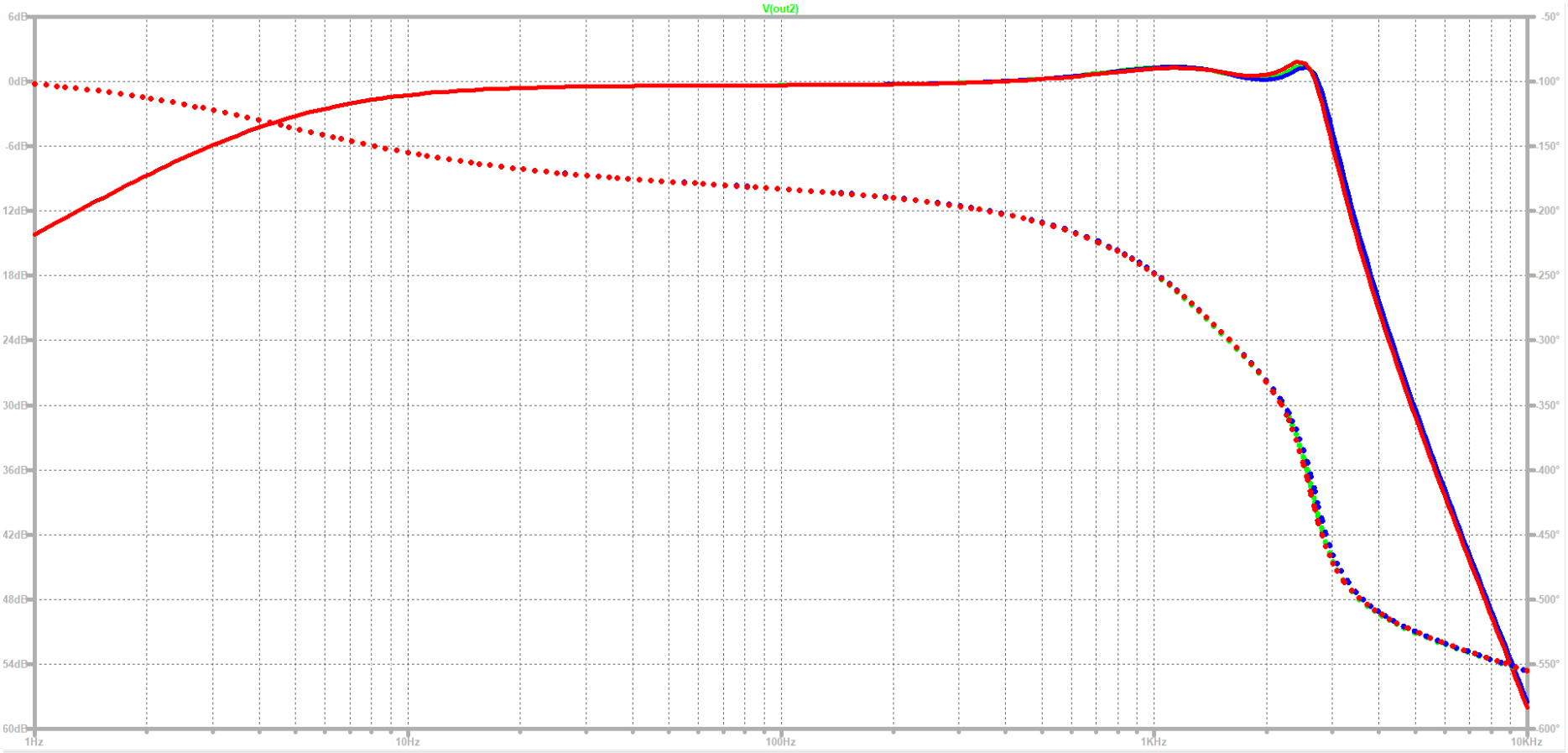


Figure 24: Input Conditioning Circuitry AC Analysis

Cut-off frequency at 2.8kHz, stopband frequency at 7.8125kHz. Maximum ripple gain of 1.5dB, unity gain through majority of passband.

7.2 Embedded Code

To operate our digital voice recorder, that is initiate recording and playback functions, or stop these functions when necessary, a state machine was implemented using embedded C code. A state machine is a structure consisting of a limited (finite) number of states, that a given system processes through in a prescribed order [15].

The state machine developed to control our digital voice recorder uses enumerated typed state definitions, controlled by a switch statement. The operating condition (recording, playing or stopped) of our state machine is defined by button presses, outlined in section 4.2 of this report. The enumerated definitions are defined as follows:

```
enum {
        DVR_STOPPED,
        DVR_RECORDING,
        DVR_PLAYING
};
```

We can produce a flow chart describing all possible processes within our state machine.

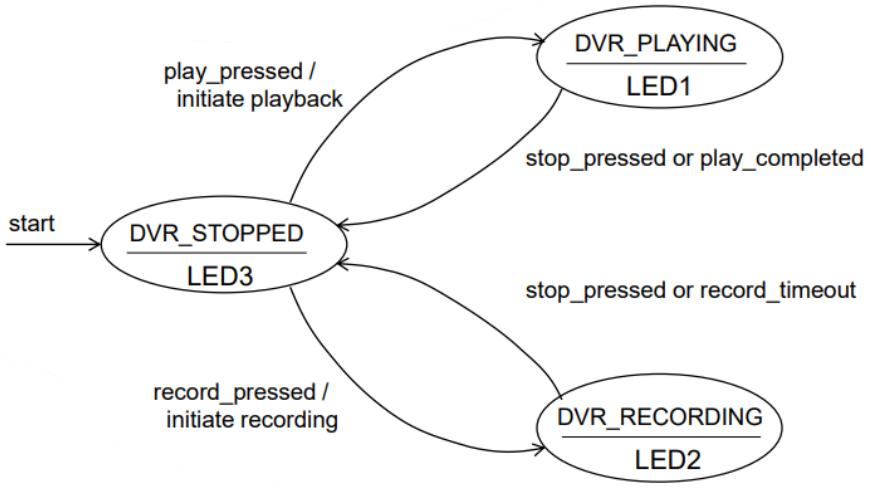


Figure 25: State Machine Flow Diagram

As we can observe in the figure above, our firmware initially configures our DVR in a stopped state, visually represented by our blue LED3. The DVR will continuously loop in this state, unless either playback or recording processes are initiated, via pushbutton S1 or S2 respectively. The state machine will then transition to one of these states corresponding to this button press and begin looping again. It will continue to loop through these stages until either pushbutton S3 is pressed, which will stop playback/recording functionality, or one of these functions' times out.

In order to write digital samples gathered by the ADC to our SD card, and then play these samples back using PWM, our DVR makes use of a circular buffer. This buffer occupies 1024 bytes of memory, and is organised into 2, 512-byte pages. Our firmware initialises this buffer on start-up, and also defines two functions, pageFull() and pageEmpty() that dictate how this buffer operates when a given page is empty/full.

7.2.1 Playback

The playback functionality initially reads two pages worth of samples into the buffer. These samples are then dequeued from the buffer, output at the same rate as they were recorded, through our PWM peripheral. Once all samples in the first page are read, the page will be marked as empty, and thus a new page of samples will be read from memory into the first page of the buffer. The buffer will now dequeue samples from the second page, until *it* is empty. New samples will be loaded into this page, and the buffer will start outputting from page one again. This process will continue until either the stop button is pressed, or we run out of samples.

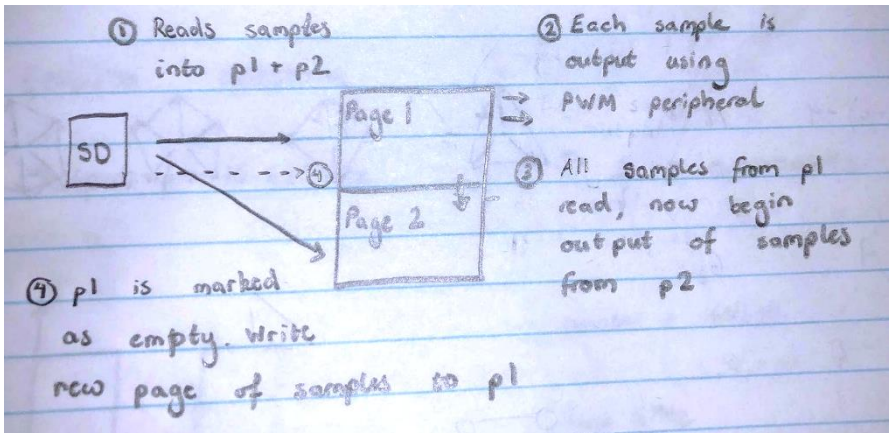


Figure 26: Firmware Playback Diagram

7.2.2 Recording

The recording functionality begins by starting the ADC. The ADC peripheral will read voltages from the Teensy board's JIN pin (PINF0) with a sampling rate of 15.625kHz (every 64µs). These voltages will then be sent to the circular buffer, starting at page 1. If a new sample is sent to the buffer every 64µs, it will take approximately 33ms to fill an entire page. Once the first page is filled with data, we write the entire page (512 samples at a time) to our SD card, and begin filling our second page. We will alternate between writing and filling each page until either the stop button is pressed, or the maximum record length of 10 seconds (305 pages) is reached.

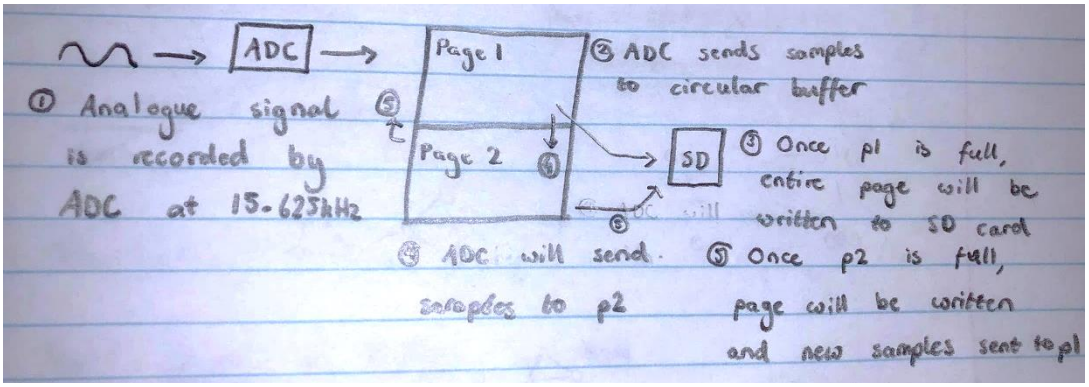


Figure 27: Firmware Recording Diagram

7.3 PWM Output

As discussed in section 2.6 of this report, we can modulate our PWM duty cycle over time to produce varying analogue values. This effect is achieved through the use of our digital carrier waveform. When playback is initiated, an interrupt service routine will execute each time this counter reaches its TOP value.

An interrupt service routine is a piece of code that is executed when a pre-defined event occurs. When this event is registered, the main program execution pauses, and control passes to the interrupt service routine. The interrupt service routine processes the interrupt and takes appropriate action, before passing control back to the main program [15].

When our TOP value overflow interrupt occurs, we will set our output compare register to the first sample stored in our buffer. This process will then remove the sample from the buffer and produce a pulse representative of our analogue waveform via PWM output.

In order to reproduce our recorded signal accurately, we will need to output each sample at the same rate as they were recorded. We know our samples are recorded at 15.625kHz, however this frequency falls within the audible spectrum perceptible to the human ear. For this reason, if we were to match our PWM frequency with this sampling frequency, we would hear our feedback from our PWM in the form of a high-pitched noise.

To prevent this from occurring, the PWM frequency was doubled by reducing our TOP (maximum counter) value by a factor of 2, to 512. In doing so, our PWM will now output at a frequency of 31.25kHz, far outside the spectrum of human hearing. Unfortunately, we are now executing our TOP value overflow interrupt, and therefore our PWM output, twice as fast, and thus our signal is played back at double the rate of its original recording.

To account for this error, each sample will now be output twice. We have defined a value that will count how many times each sample has been played. Every time our TOP overflow interrupt occurs, we will check is value. If the sample has not been played, our compare register will be set to the next sample value in our buffer, the sample will be dequeued and our counter variable will increment. If the sample has been played once, our compare register value will remain the same, and the next sample in the buffer will not be dequeued. We will reset our counter value such that we can output the next sample in the buffer.

8 Implementation

With our components selected, circuitry simulated, and firmware accounted for, we can now begin implementing our technical design decisions in order to produce our final, digital voice recorder. Our analogue input circuitry will be produced on a breadboard and interfaced with our Teensy development board, and our firmware will be loaded onto our ATMega34U4 microcontroller using QUTs Teensy loader.

Figure 28: QUT Teensy Loader

8.1 Bill of Materials

Designator	Value	Device	Description	Qty	Package
U1, U2, U3	LMC6484	LMC6484	IC, Quad CMOS Operational-Amplfier	1	DIP-14
R1	8.2K	RESISTOR-AXIALHORIZONTAL	Resistor, Axial, Metal Film, 0.5W	1	AXIAL-P10.16
R2	16K	RESISTOR-AXIALHORIZONTAL	Resistor, Axial, Metal Film, 0.5W	1	AXIAL-P10.16
R3	5.1K	RESISTOR-AXIALHORIZONTAL	Resistor, Axial, Metal Film, 0.5W	1	AXIAL-P10.16
R4, R5	10K	RESISTOR-AXIALHORIZONTAL	Resistor, Axial, Metal Film, 0.5W	2	AXIAL-P10.16
R6	1.2MEG	RESISTOR-AXIALHORIZONTAL	Resistor, Axial, Metal Film, 0.5W	1	AXIAL-P10.16
R7, R8	20K	RESISTOR-AXIALHORIZONTAL	Resistor, Axial, Metal Film, 0.5W	2	AXIAL-P10.16
C1	5.6nF	CAPCERAMIC-DISC	Capacitor, Radial, Non-polarised	1	CAP-DISC-P5.08
C2	18nF	CAPCERAMIC-DISC	Capacitor, Radial, Non-polarised	1	CAP-DISC-P5.08
C3	1nF	CAPCERAMIC-DISC	Capacitor, Radial, Non-polarised	1	CAP-DISC-P5.08
C4	68nF	CAPCERAMIC-DISC	Capacitor, Radial, Non-polarised	1	CAP-DISC-P5.08
C5	3.3uF	CAP-POLRB-D5.0XP2.0	Capacitor, Radial, Polarised	1	CAP-RB-P2.0-D5.0
M1	CMA-6542TF-K	MIC-COND-ANALOG-OMNI-42DB	Omnidirectional Electret Condenser Microphone	1	

Designator	Manufacturer	MPN	Supplier	SKU	MOQ	Price
U1, U2, U3	Texas Instruments	LMC6484AIN	Jaycar	ZL-3484	1	$9.95
R1, R9	Generic	8.2k Ohm 0.5 Watt Metal Film Resistors	Jaycar	RR-0594	1	$0.07
R2	Generic	16k Ohm 0.5 Watt Metal Film Resistors	Jaycar	RR-0601	1	$0.07
R3	Generic	5.1k Ohm 0.5 Watt Metal Film Resistors	Jaycar	RR-0589	1	$0.07
R4, R5	Generic	10k Ohm 0.5 Watt Metal Film Resistors	Jaycar	RR-0596	2	$0.14
R6	Yageo	10M Ohm 0.25 Watt Metal Film Resistors	Mouser	25FT-52-1M2	1	$0.38
R7, R8	Generic	20k Ohm 0.5 Watt Metal Film Resistors	Jaycar	RR-0603	2	$0.14
C1	Generic	5.6nF 100VDC Polyester Capacitor	Jaycar	RG-5050	1	$0.30
C2	Generic	18nF 100VDC Polyester Capacitor	Jaycar	RG-5080	1	$0.30
C3	Generic	1nF 50VDC Ceramic Capacitors	Jaycar	RC-5336	1	$0.16
C4	Generic	68nF 100VDC Polyester Capacitor	Jaycar	RG-5115	1	$0.30
C5	Generic	3.3uF 63VDC Electrolytic RB Capacitor	Jaycar	RE-6045	1	$0.30
M1	CUI	Analog Microphone Electret Condenser 1.5V ~ 10V	Digi-Key	102-1719-ND	1	

Table 1: Bill of Materials (BoM)

8.2 Assembled Prototype

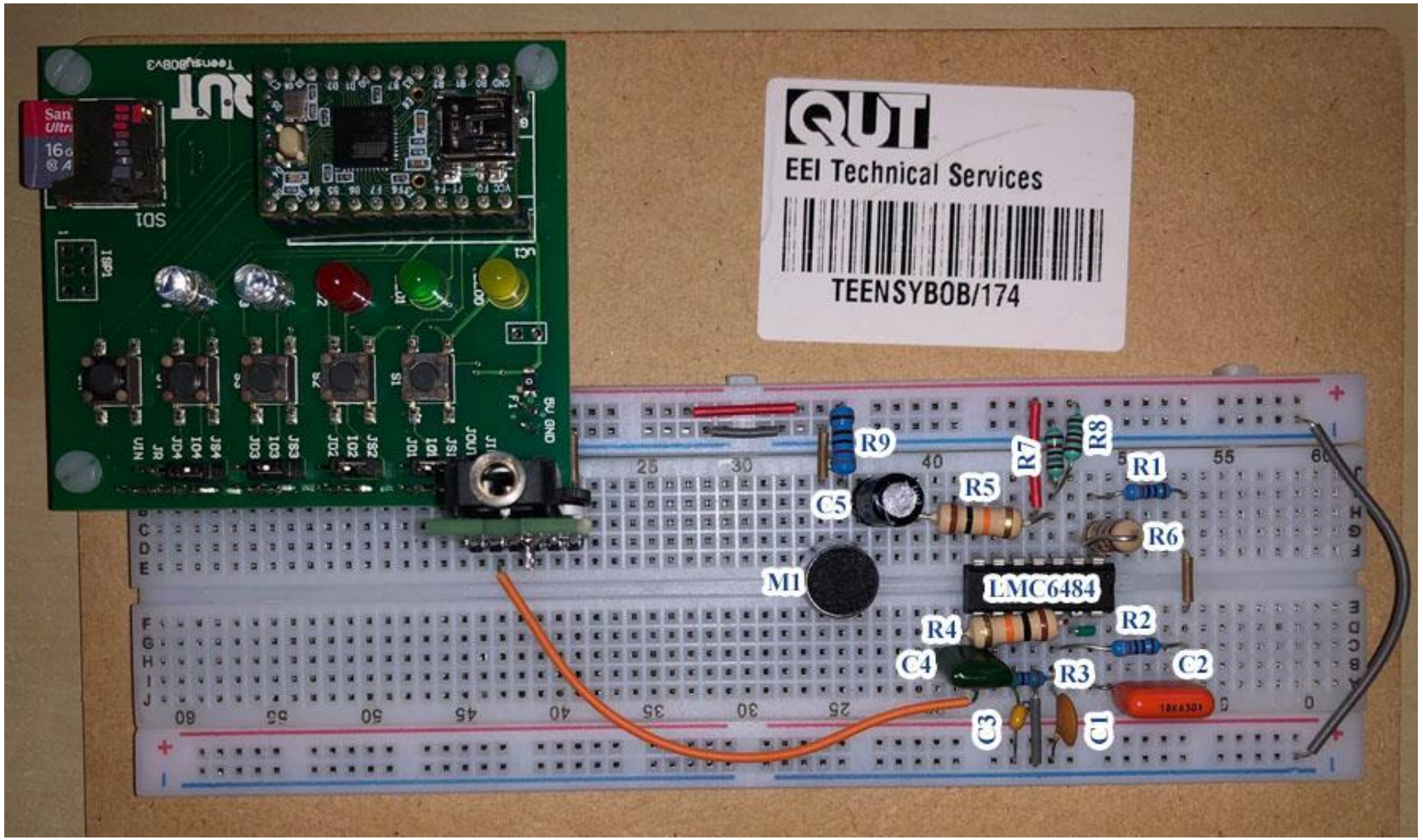


Figure 29: Assembled Input Conditioning Prototype Construction w/ Annotations

Power Supplied to Circuit by Teensy USB Power Supply (5V)

8.3 Source Code

A full copy of the main.c source code is provided in this report as an appendix (4). The primary functions within this code are outlined below:

8.3.1 Global Variables

```
uint16_t pageCount = 0;  // Page counter - used to terminate recording
uint16_t newPage = 0;    // Flag that indicates a new page is available for read/write
uint8_t stop = 0;        // Flag that indicates playback/recording is complete
uint8_t play = 0;        // Flag that indicates playback is in operation
uint32_t tsamples = 0;   // Variable to store total samples
```

8.3.2 Initialisation

```
void firmware_init(){
        OCR1A = 512.0;          // TOP, 31.25kHz
        TCCR1A = 0b00100011;    // Fast PWN (TOP = OCR1A), set OCR1B on TOP, reset on CMP
        TCCR1B = 0b00011001;    // Fast PWM (TOP = OCR1A), /1 pre-scalar
        TIMSK1 = 0b00000001;    // Enables overflow interrupt for timer1

        DDRB |= (1<<PINB6);     // Allows output to JOUT
        // Turns on output for LED1, 2 & 3
        DDRD |= (1<<PIND4);     // LED 1
        DDRD |= (1<<PIND5);     // LED 2
        DDRD |= (1<<PIND6);     // LED 3
}
```

8.3.3 Overflow Interrupt

```
volatile uint8_t dbl = 0;       // Define variable to play each sample twice
/**
 * ISR: Timer1 Overflow
 *
 * Interrupt service routine which executes on timer 1 overflow.
 */
ISR(TIMER1_OVF_vect) {
                // If playback is running
                if (play && dbl == 0){
                        // If sample has not been replayed
                        OCR1B = buffer_dequeue();       // Output a sample from the buffer
                        dbl = 1;                        // Increment playback counter
                }
                else {
                        // Play sample again
                        dbl = 0;                        // Reset playback counter
                }
}
```

8.3.4 Full Page

```
void pageFull() {
        if(!(--pageCount)) {
                // If all pages have been read
                adc_stop();     // Stop recording (disable new ADC conversions)
                stop = 1;       // Flag recording complete
        } else {
                newPage = 1;    // Flag new page is ready to write to SD card
        }
}
```

8.3.5 Empty Page

```
void pageEmpty() {
        if (tsamples == 0 && play){
                // If all samples have been played
                stop = 1;       // Stop playback
        } else {
                tsamples -= 512;                        // Remove previous page of samples
                wave_read(buffer_writePage(), 512);     // Write new page of samples to the buffer
        }
}
```

8.3.6 Record

```
void dvr_record() {
        buffer_reset();         // Reset buffer state

        pageCount = 305;// Maximum record time of 10 sec
        newPage = 0;            // Clear new page flag

        wave_create();          // Create new wave file on the SD card
        adc_start();            // Begin sampling

        PORTD |= (1<<PIND5);            // LED2 = ON
        PORTD &= ~(1<<PIND6);           // LED3 = OFF
}
```

8.3.7 Playback

```
void dvr_playback(){
        buffer_reset();         // Reset buffer state

        tsamples = wave_open();  // Open the wave file stored on the SD card/store it's samples

        wave_read(buffer_writePage(), 1024);     // Read first two pages of samples into the buffer

        PORTD |= (1<<PIND4);    // LED1 = ON
        PORTD &= ~(1<<PIND6);   // LED3 = OFF
}
```

8.3.8 Main Loop

```
// Loop forever (state machine)
for(;;) {

        // Define state of switch
        s1 = (1<<PINF4)&PINF;   // Switch 1
        s2 = (1<<PINF5)&PINF;   // Switch 2
        s3 = (1<<PINF6)&PINF;   // Switch 3

        // Switch depending on state
        switch (state) {
                case DVR_STOPPED:
                        PORTD &= ~(1<<PIND4);           // LED1 = OFF
                        PORTD &= ~(1<<PIND5);           // LED2 = OFF
                        PORTD |= (1<<PIND6);            // LED3 = ON

                        // Switch 2 pressed
                        if (!s2) {
                                printf("Recording...");  // Output status to console
                                dvr_record();           // Initiate recording
                                state = DVR_RECORDING;  // Transition to "recording" state
                        }
```

```
                        // Switch 1 pressed
                        if (!s1 && !play) {
                                printf("Playing...");    // Output status to console
                                dvr_playback();          // Initiate recording
                                state = DVR_PLAYING;     // Transition to "playing" state
                        }

                                play = 0;                       // Define play state
                        break;
                case DVR_RECORDING:
                        // Switch 3 Pressed
                        if (!s3) {
                                pageCount = 1;  // Finish recording last page
                        }

                        // Write samples to SD card when buffer page is full
                        if (newPage) {
                                newPage = 0;    // Acknowledge new page flag
                                wave_write(buffer_readPage(), 512);
                        } else if (stop) {
                                // Stop is flagged when the last page has been recorded
                                stop = 0;                               // Acknowledge stop flag
                                wave_write(buffer_readPage(), 512);     // Write final page
                                wave_close();                           // Finalise WAVE file
                                printf("DONE!\n");                      // Print status to console
                                state = DVR_STOPPED;                    // Trans to stopped state
                        }
                        break;
                case DVR_PLAYING:
                        // Switch 3 Pressed
                        if (!s3) {
                                stop = 0;                       // Acknowledge stop flag
                                printf("DONE!\n");              // Print status to console
                                state = DVR_STOPPED;            // Transition to stopped state
                        }
                        if (!stop){
                                play = 1;                       // Define play state
                        }
                        else {
                                stop = 0;                       // Acknowledge stop flag
                                printf("DONE!\n");              // Print status to console
                                state = DVR_STOPPED;            // Transition to stopped state
                        }
                        break;
                default:
                        // Invalid state, return to valid idle state (stopped)
                        printf("ERROR: State machine in main entered invalid state!\n");
                        state = DVR_STOPPED;
                        break;

        } // END switch(state)

} // END for(;;)
```

9 Experimental Results

To validate the implementation of our technical design, two methods for gathering experimental results were used, based on which module of the DVR we are testing. To test our input conditioning circuitry, an oscilloscope and signal generator were utilised, to measure the response of the circuit at different frequencies. When testing our embedded code, debugging techniques were employed, to ensure our circuit was behaving as intended.

9.1 Microphone Bias/Amplification, 2kHz

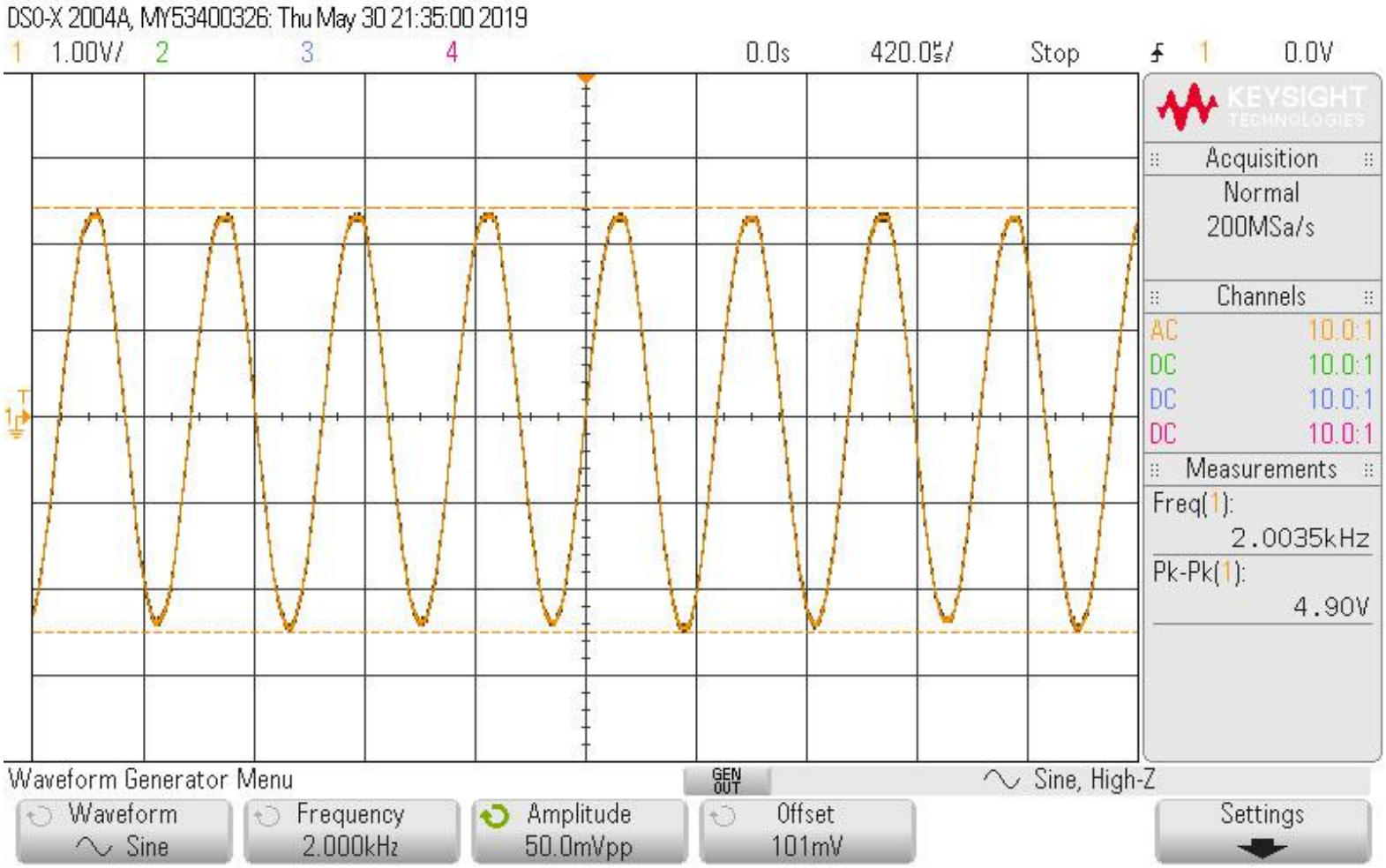


Figure 30: Microphone Bias & Amplification at 2kHz

Microphone signal modelled using signal generator at 50mVpp. Frequency selected based on important frequency range in our human speech frequency bandwidth. Signal has been amplified to 4.9V, just below the 5V maximum, with very little frequency deviation (>0.2%), and biased around 2.5V

9.2 Microphone Bias/Amplification, 20kHz

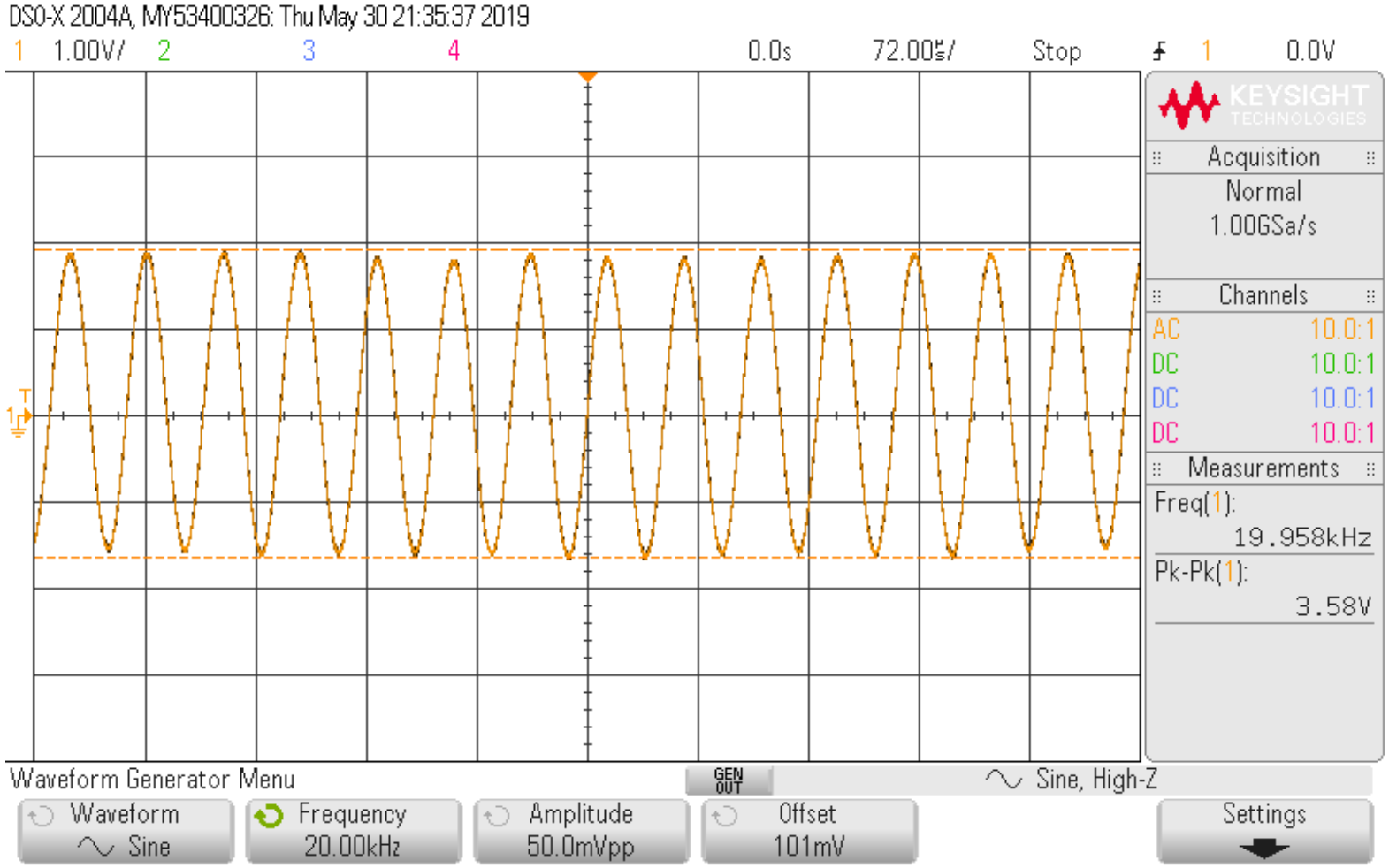


Figure 31: Microphone Bias & Amplification at 2kHz

Microphone signal modelled using signal generator at 50mVpp. Compared to our amplification in figure 9.1, our signal is now only registering a pk-pk voltage of 3.58V. This effect is a result of the operational amplifiers gain-bandwidth product, discussed in section 7.1.2. Our signal is still biased at 2.5V.

9.3 Filter Frequency Response

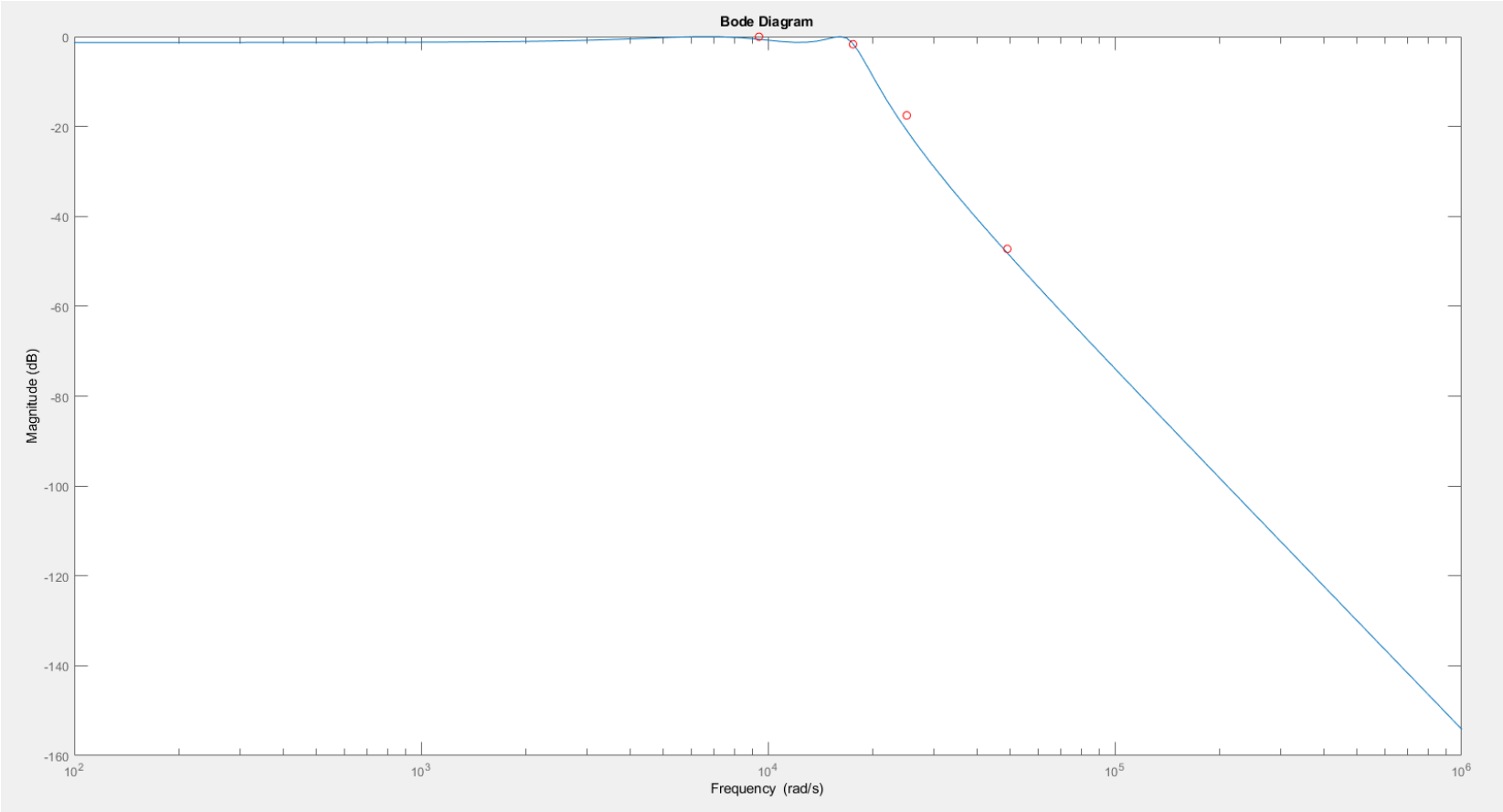


Figure 32: Experimental Data Plot Against Modelled Filter Bode Diagram

Experimental data gathered and plot against the frequency response of our filter design modelled in MATLAB. Experimental data gathered in Appendix 5. Data represents -0.02 gain at 1700Hz, -1.78 gain at 2800Hz, -17.58 gain at 4000Hz and -47.23 gain at 7800Hz.

As we can observe in figures 30-32 above, our experimental data matches our designed specifications quite closely. When modelled at 50mVpp, our microphone signal outputs at approximately 4.9V, just slightly below our 5V limit, and is biased around 2.5V. Our Chebyshev filter begins to attenuate frequencies just outside of our passband, at 2800Hz, and reaches our stopband point at approximately 7800Hz. Roll-off is also consistent with our modelled frequency response.

9.4 Embedded Code

Stop State | Record State | Playback State

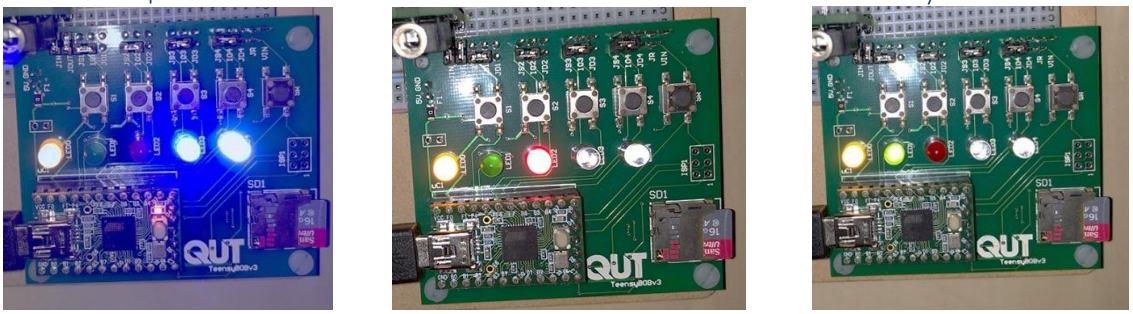


Figure 33: Stop, Record and Playback States

As we can observe in the figure above, each state is visually represented by an LED, and can be initiated through push button presses. By printing commands to our terminal, based on the intended operation of our functions, we can ensure they are working as intended.

9.4.1 Recording

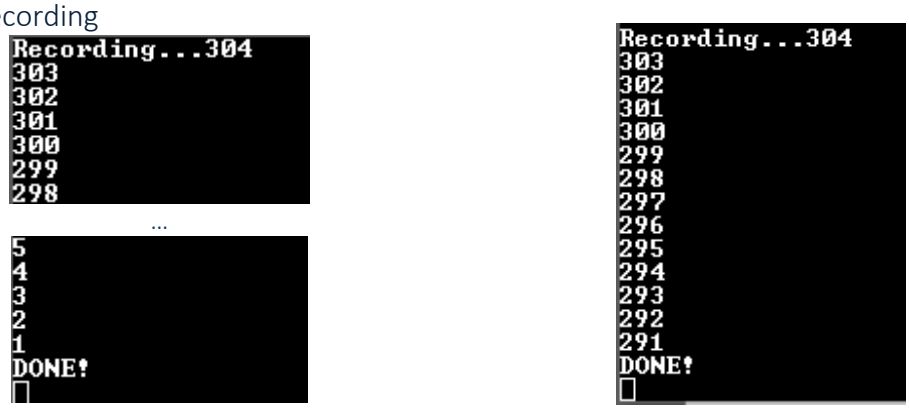


Figure 34: pageCount variable Behaviour During Record Stage

Left displays behaviour when record button is pressed and the maximum record length is reached. The variable will count down from page 305 (10 seconds of samples) to 0, and then end the recording (DONE message produced). Right displays behaviour when the stop button is pressed. The variable will begin counting down, and end when the stop button is pressed.

9.4.2 Playback

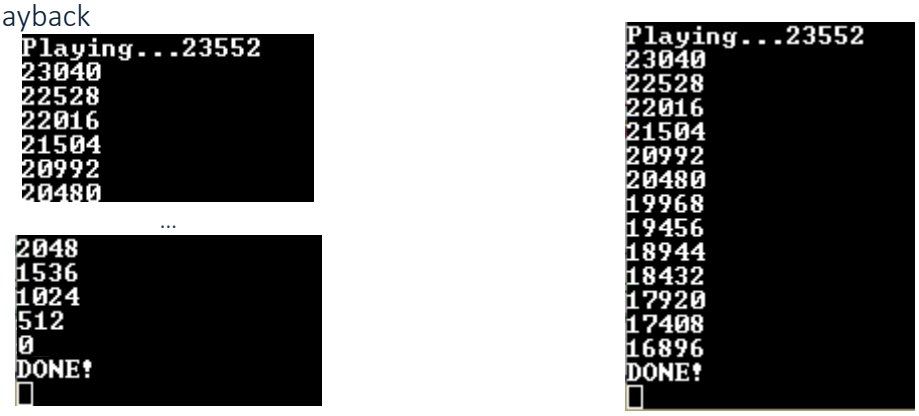


Figure 35: tSamples variable Behaviour During Playback Stage

Left displays behaviour when playback button is pressed and all samples are output. The variable will count down from the total samples recorded to 0, and then end the playback (DONE message produced). Right displays behaviour when the stop button is pressed. The variable will begin counting down, and end when the stop button is pressed.

10 Future Work

It can be said that, after rigorous analysis, design, prototyping, and testing, our final digital voice recorder meets all our specified requirements. Our input signal is conditioned such that is free from distortion, amplified around the centre of our maximum voltage, all while preserving frequency content within the human speech frequency bandwidth. Our firmware handles the operation of the digital voice recorder exactly to specification, and produces a PWM output that reproduces our input signal to a very high quality.

With that said, there will always be areas of improvement, or enhancements we could make to our design to improve its overall quality. In the case of this digital voice recorder design, many improvements can be made based on aspects or inclusions that were considered to be outside of our project scope, as outlined in section 5 of this report.

An LCD display could be interfaced with our microcontroller, such that we can provide an additional layer of feedback to the user. This LCD screen would provide key sound information related to our signal such as levels or frequency bandwidth. This panel would also allow for the streamlined management of multiple signal recordings, and thus we would no longer have to overwrite our wave file each time we record a new one.

Signal quality could be increased through either interpolation techniques, used to smooth our input signal by creating new samples in between each original, or by increasing our ADC sample rate. Increasing the ADC sample rate, in conjunction with a more rigorous filter design selection, may open up the possibility of utilising this recording device for frequencies beyond the human speech frequency bandwidth. With these improvements, we may be able turn our digital voice recorder, into an outright microphone recording device

References

[1] E. B. Brixen, “Facts About Speech Intelligibility”, *dpamicrophones.com*, para. 2-4, Jan. 20, 2016. [Online]. Available https://www.dpamicrophones.com/mic-university/facts-about-speech-intelligibility [Accessed: May. 23, 2019].

[2] B. Munir, “Voice Fundamentals – Human Speech Frequency”, *unifiedoverip.com*, para. 4, Mar. 10, 2012. [Online]. Available http://www.uoverip.com/voice-fundamentals-human-speech-frequency/ [Accessed: May. 23, 2019].

[3] Prof. C. E. Efstathiou, “Signal Sampling: Nyquist - Shannon Theorem”, *National and Kapodistrian University of Athens, Department of Chemistry*, n.d. [Online]. Available http://195.134.76.37/applets/AppletNyquist/Appl_Nyquist2.html [Accessed: May. 23, 2019].

[4] Tech Target, “Nyquist Theorem”, *whatis.com*, para. 3, Sep. 2005. [Online]. Available https://whatis.techtarget.com/definition/Nyquist-Theorem [Accessed: May. 23, 2019].

[5] Aspencore, “Second Order Filters”, *electronics-tutorials.com*, para. 1, n.d. [Online]. Available https://www.electronics-tutorials.ws/filter/second-order-filters.html [Accessed: May. 23, 2019].

[6] EEEGuide, “Higher Order Filter Design”, *EEEGuide.com*, para. 2, Aug. 20, 2012. [Online]. Available http://www.eeeguide.com/higher-order-filter-design/ [Accessed: May. 23, 2019].

[7] B. Murmann, “EE315A - VLSI Signal Conditioning Circuits”, *Stanford University*, chapter 2, 2010. [Online]. Available Archive [Accessed: May. 23, 2019].

[8] Dr. M. Broadmeadow. EGB240. Class Lecture, Topic: “Filter Design” Science and Engineering Faculty, Queensland University of Technology, Brisbane QLD, Apr. 8, 2019.

[9] Walsin Technology Corporation, “Product Catalogue”, *passivecomponents.com*, Aug, 2018. [Online] Available http://www.passivecomponent.com/wp-content/uploads/2018/10/chipR.pdf [Accessed: May. 24, 2019].

[10] Dr. M. Broadmeadow. EGB240. Class Lecture, Topic: “Filter Tuning” Science and Engineering Faculty, Queensland University of Technology, Brisbane QLD, Apr. 15, 2019.

[11] T, Hirzel. “PWM”, *arduino.com*, para. 1, n.d. [Online]. Available https://www.arduino.cc/en/Tutorial/PWM [Accessed: May. 24, 2019].

[12] Dr. M. Broadmeadow. EGB240. Class Lecture, Topic: “Intro to Assessments 2 & 3, Sampling & PWM” Science and Engineering Faculty, Queensland University of Technology, Brisbane QLD, Mar. 25, 2019.

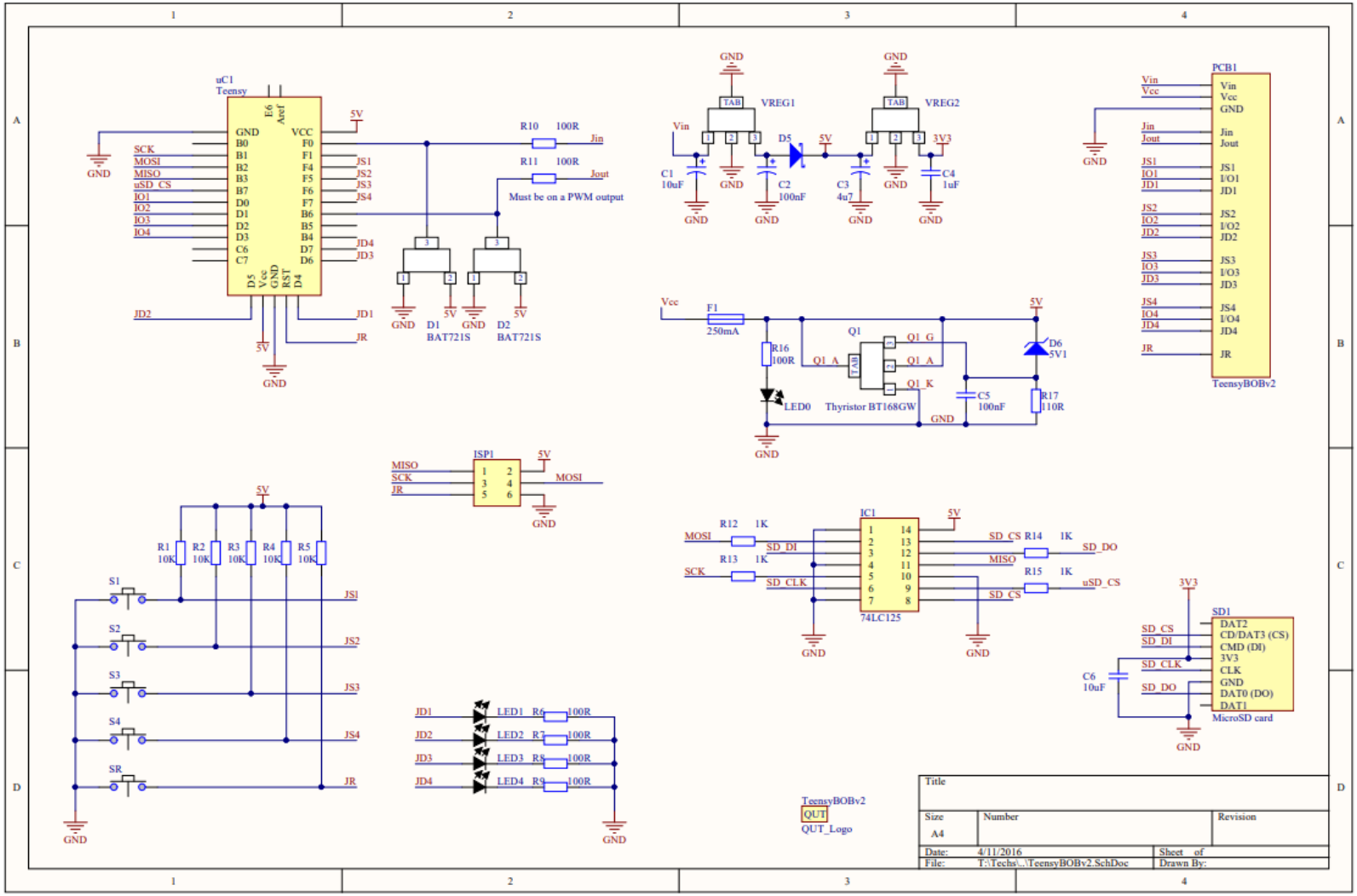
[13] W, Maxwell, M. Cacan and C. Haile, “Pulse Width Modulation”, *The George w. Woodruff School of Mechanical Engineering, Georgia Tech*, 2013. [Online] Available http://ume.gatech.edu/mechatronics_course/PWM_Fall13.pptx [Accessed: May. 24, 2019].

[14] Chapter 18/Lesson 14, “Interpolation in Statistics: Definition, Formula & Example”, *study.com*, n.d. [Online]. Available https://study.com/academy/lesson/interpolation-in-statistics-definition-formula-example.html [Accessed: May. 24, 2019].

[15] Dr. M. Broadmeadow. EGB240. Class Lecture, Topic: “Real-time Digital Embedded Systems” Science and Engineering Faculty, Queensland University of Technology, Brisbane QLD, May. 13, 2019.

Appendices

1 Teensy Development Board



2 Microphone Specifications

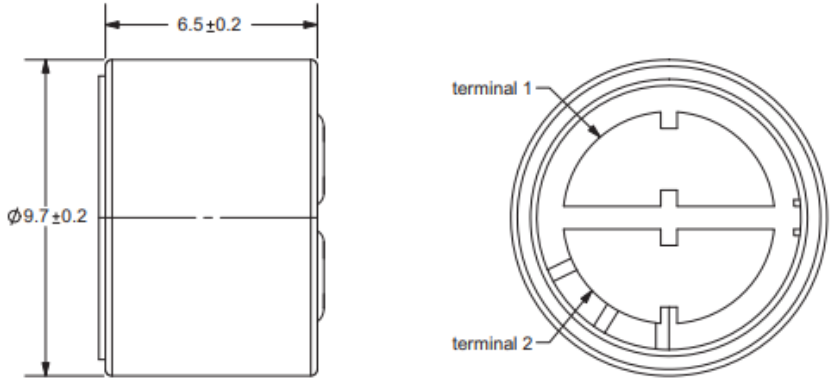
SPECIFICATIONS

directivity	omnidirectional	
sensitivity (S)	-42 ±2 dB	f = 1KHz, 1Pa 0dB = 1V/Pa
sensitivity reduction (ΔS-Vs)	-3 dB	f = 1KHz, 1Pa Vs = 1.5 ~ 1.0 V dc
operating voltage	1.5 V dc (standard), 10 V dc (max.)	
output impedance (Zout)	1.5 KΩ	f = 1KHz, 1Pa
operating frequency (f)	100 ~ 20,000 Hz	
current consumption (IDSS)	0.5 mA max.	Vs = 1.5 V dc RL = 1.5KΩ
signal to noise ratio (S/N)	58 dBA	f = 1KHz, 1Pa A-weighted
operating temperature	-20 ~ +70° C	
storage temperature	-20 ~ +70° C	
dimensions	ø9.7 x 6.5 mm	
weight	0.85 g max.	
material	Al	
terminal	terminal type (hand soldering only)	
RoHS	yes	

note: We use the "Pascal (Pa)" indication of sensitivity as per the recomendation of I.E.C. (International Electrotechnical Commission). The sensitivity of "Pa" will increase 20dB compared to the "ubar" indication. Example: -60dB (0dB = 1V/ubar) = -40dB (1V/Pa)

APPEARANCE DRAWING

tolerances not shown: ±0.3mm



3 Filter Tuning

	A	B	C	D	E	F	G
1	Q	0.8287					
2	C1	5.6	nF				
3							
4	m	R ratio	n	C2			
5	0	1.00	2.75	15.38			
6	1	1.10	2.75	15.42			
7	2	1.21	2.77	15.53			
8	3	1.33	2.80	15.70			
9	4	1.47	2.85	15.96			
10	5	1.62	2.91	16.29			
11	6	1.78	2.98	16.69			
12	7	1.96	3.07	17.18			
13	8	2.15	3.17	17.76			
14	9	2.37	3.29	18.43			
15	10	2.61	3.43	19.20			
16	11	2.87	3.59	20.08			
17	12	3.16	3.76	21.07			
18	13	3.48	3.96	22.18			
19	14	3.83	4.18	23.43			
20	15	4.22	4.43	24.82			
21	16	4.64	4.71	26.37			
22	17	5.11	5.02	28.09			
23	18	5.62	5.36	30.00			
24	19	6.19	5.74	32.12			
25	20	6.81	6.15	34.46			
26	24	10.00	8.31	46.53			
27							
28							
29	wn	8.83E+03	rad/s		C1	5.6	nF
30	m	1.96			C2	18	nF
31							
32	R1	8060	Ohm				
33	R2	15798					

	A	B	C	D	E	F	G
1	Q	3.8842					
2	C1	1	nF				
3							
4	m	R ratio	n	C2			
5	0	1.00	60.35	60.35			
6	1	1.10	60.49	60.49			
7	2	1.21	60.91	60.91			
8	3	1.33	61.61	61.61			
9	4	1.47	62.60	62.60			
10	5	1.62	63.89	63.89			
11	6	1.78	65.49	65.49			
12	7	1.96	67.41	67.41			
13	8	2.15	69.68	69.68			
14	9	2.37	72.31	72.31			
15	10	2.61	75.33	75.33			
16	11	2.87	78.77	78.77			
17	12	3.16	82.65	82.65			
18	13	3.48	87.02	87.02			
19	14	3.83	91.91	91.91			
20	15	4.22	97.37	97.37			
21	16	4.64	103.45	103.45			
22	17	5.11	110.21	110.21			
23	18	5.62	117.70	117.70			
24	19	6.19	125.99	125.99			
25	20	6.81	135.18	135.18			
26	24	10.00	182.55	182.55			
27							
28							
29	wn	1.71E+04	rad/s		C1	1	nF
30	m	1.96			C2	68	nF
31							
32	R3	5055	Ohm				
33	R4	9908					

4 Source Code

```
/**
 * main.c - EGB240 Digital Voice Recorder Skeleton Code
 *
 * This code provides a skeleton implementation of a digital voice
 * recorder using the Teensy microcontroller and QUT TensyBOBv2
 * development boards. This skeleton code demonstrates usage of
 * the EGB240DVR library, which provides functions for recording
 * audio samples from the ADC, storing samples temporarily in a
 * circular buffer, and reading/writing samples to/from flash
 * memory on an SD card (using the FAT file system and WAVE file
 * format.
 *
 * This skeleton code provides a recording implementation which
 * samples CH0 of the ADC at 8-bit, 15.625kHz. Samples are stored
 * in flash memory on an SD card in the WAVE file format. The
 * filename is set to "EGB240.WAV". The SD card must be formatted
 * with the FAT file system. Recorded WAVE files are playable on
 * a computer.
 *
 * LED4 on the TeensyBOBv2 is configured to flash as an
 * indicator that the programme is running; a 1 Hz, 50 % duty
 * cycle flash should be observed under normal operation.
 *
 * A serial USB interface is provided as a secondary control and
 * debugging interface. Errors will be printed to this interface.
 *
 * Version: v1.0
 *    Date: 10/04/2016
 *  Author: Mark Broadmeadow
 *  E-mail: mark.broadmeadow@qut.edu.au
 */

 /*****************************************************************************/
 /* INCLUDED LIBRARIES/HEADER FILES                                           */
 /*****************************************************************************/
#include <avr/io.h>
#include <avr/interrupt.h>

#include <stdio.h>

#include "serial.h"
#include "timer.h"
#include "wave.h"
#include "buffer.h"
#include "adc.h"

/*****************************************************************************/
/* ENUM DEFINITIONS                                                          */
/*****************************************************************************/
enum {
        DVR_STOPPED,
        DVR_RECORDING,
        DVR_PLAYING
};

/*****************************************************************************/
/* GLOBAL VARIABLES                                                          */
/*****************************************************************************/
uint16_t pageCount = 0;  // Page counter - used to terminate recording
uint16_t newPage = 0;    // Flag that indicates a new page is available for read/write
uint8_t stop = 0;                // Flag that indicates playback/recording is complete
uint8_t play = 0;                // Flag that indicates playback is in operation
uint32_t tsamples = 0;   // Variable to store total samples

/*****************************************************************************/
/* FUNCTION PROTOTYPES                                                       */
/*****************************************************************************/
void pageFull();
void pageEmpty();
```

```
/*****************************************************************/
/* INITIALISATION FUNCTIONS                                      */
/*****************************************************************/

// Initialise PLL (required by USB serial interface, PWM)
void pll_init() {
        PLLFRQ = 0x6A; // PLL = 96 MHz, USB = 48 MHz, TIM4 = 64 MHz
}

// Configure system clock for 16 MHz
void clock_init() {
        CLKPR = 0x80;    // Prescaler change enable
        CLKPR = 0x00;    // Prescaler /1, 16 MHz
}

void milestone_init(){
        OCR1A = 1023.0;          // TOP, 15.625kHz
        //OCR1A = 62500;         // TOP, 1Hz
        OCR1B = OCR1A * 0.5;     // 50% duty cycle
        TCCR1A = 0b00100011;     // Fast PWN (TOP = OCR1A), set OCR1B on TOP, reset on CMP
        TCCR1B = 0b00011001;     // Fast PWM (TOP = OCR1A), /1 pre-scalar
        //TCCR1B = 0b00011100;   // Fast PWM (TOP = OCR1A), /256 pre-scalar

        DDRB |= (1<<PINB6);      // Allows output to JOUT
        // Turns on output for LED1, 2 & 3
        DDRD |= (1<<PIND4); // LED 1
        DDRD |= (1<<PIND5); // LED 2
        DDRD |= (1<<PIND6); // LED 3
        //PORTB |= (1<<PINB6);
}

void firmware_init(){
        OCR1A = 512.0;           // TOP, 31.25kHz
        TCCR1A = 0b00100011;     // Fast PWN (TOP = OCR1A), set OCR1B on TOP, reset on CMP
        TCCR1B = 0b00011001;     // Fast PWM (TOP = OCR1A), /1 pre-scalar
        TIMSK1 = 0b00000001;     // Enables overflow interrupt for timer1

        DDRB |= (1<<PINB6);      // Allows output to JOUT
        // Turns on output for LED1, 2 & 3
        DDRD |= (1<<PIND4); // LED 1
        DDRD |= (1<<PIND5); // LED 2
        DDRD |= (1<<PIND6); // LED 3
}

// Initialise DVR subsystems and enable interrupts
void init() {
        cli();                   // Disable interrupts
        clock_init();    // Configure clocks
        pll_init();     // Configure PLL (used by Timer4 and USB serial)
        serial_init();   // Initialise USB serial interface (debug)
        timer_init();    // Initialise timer (used by FatFs library)
        buffer_init(pageFull, pageEmpty);  // Initialise circular buffer (must specify callback
functions)
        adc_init();              // Initialise ADC
        firmware_init();// Initialise peripherals
        sei();                   // Enable interrupts
        // Must be called after interrupts are enabled
        wave_init();     // Initialise WAVE file interface
}

/*****************************************************************/
/* CALLBACK FUNCTIONS FOR CIRCULAR BUFFER                        */
/*****************************************************************/

// CALLED FROM BUFFER MODULE WHEN A PAGE IS FILLED WITH RECORDED SAMPLES
void pageFull() {
        if(!(--pageCount)) {
                // If all pages have been read
                adc_stop();              // Stop recording (disable new ADC conversions)
                stop = 1;                // Flag recording complete
        } else {
                newPage = 1;     // Flag new page is ready to write to SD card
        }
}
```

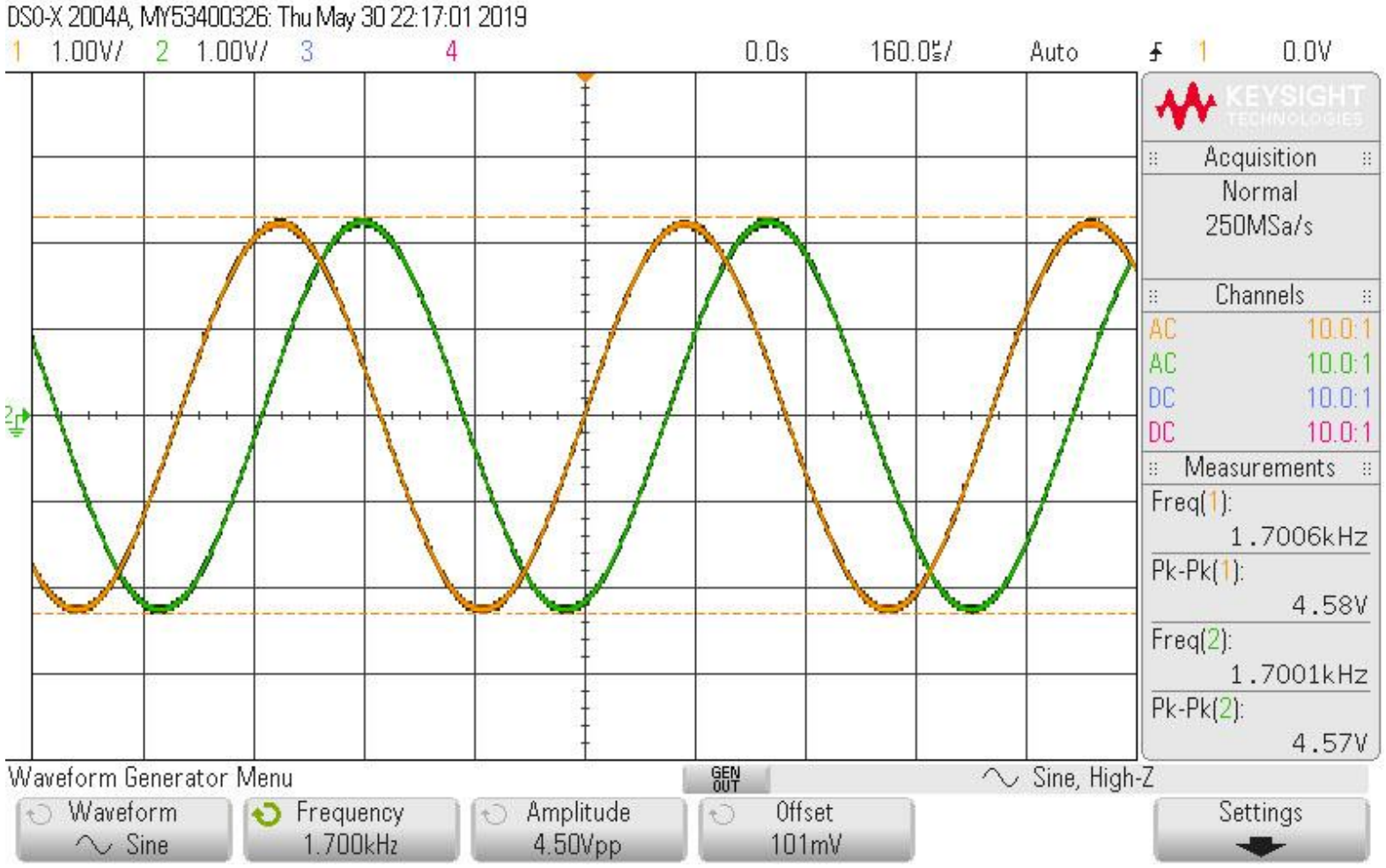
```
// CALLED FROM BUFFER MODULE WHEN A NEW PAGE HAS BEEN EMPTIED
void pageEmpty() {
        // TODO: Implement code to handle "page empty" callback
        if (tsamples == 0 && play){
                // If all samples have been played
                stop = 1;       // Stop playback
        } else {
                tsamples -= 512;                                // Remove previous page of samples
                wave_read(buffer_writePage(), 512);     // Write new page of samples to the buffer
                //printf("%lu\n", tsamples);
        }
}

/*****************************************************************/
/* RECORD/PLAYBACK ROUTINES                                      */
/*****************************************************************/

// Initiates a record cycle
void dvr_record() {
        buffer_reset();         // Reset buffer state

        pageCount = 305;// Maximum record time of 10 sec
        newPage = 0;            // Clear new page flag

        wave_create();          // Create new wave file on the SD card
        adc_start();            // Begin sampling

        // TODO: Add code to handle LEDs
        PORTD |= (1<<PIND5);            // LED2 = ON
        PORTD &= ~(1<<PIND6);           // LED3 = OFF
}

// TODO: Implement code to initiate playback and to stop recording/playback.
void dvr_playback(){
        buffer_reset();         // Reset buffer state

        tsamples = wave_open();   // Open the wave file stored on the SD card and store it's samples
        //printf("%lu\n", tsamples);

        wave_read(buffer_writePage(), 1024);     // Read the first two pages of samples into the
buffer

        PORTD |= (1<<PIND4);            // LED1 = ON
        PORTD &= ~(1<<PIND6);           // LED3 = OFF
}

volatile uint8_t dbl = 0;       // Define variable to play each sample twice
/**
 * ISR: Timer1 Overflow
 *
 * Interrupt service routine which executes on timer 1 overflow.
 */
ISR(TIMER1_OVF_vect) {
                // If playback is running
                if (play && dbl == 0){
                        // If sample has not been replayed
                        OCR1B = buffer_dequeue();       // Output a sample from the buffer
                        dbl = 1;                                // Increment playback counter
                        // Debugging
                        //printf("%i\n", dbl);
                        //printf("%u\n", OCR1B);
                }
                else {
                        // Play sample again
                        dbl = 0;                                // Reset playback counter
                }
}
```

```
/*****************************************************************/
/* MAIN LOOP (CODE ENTRY)                                        */
/*****************************************************************/
int main(void) {
        uint8_t state = DVR_STOPPED;     // Start DVR in stopped state
        uint8_t s1, s2, s3;                              // Define variables for switch presses

        // Initialisation
        init();

        // Loop forever (state machine)
        for(;;) {

                // Define state of switch
                s1 = (1<<PINF4)&PINF;    // Switch 1
                s2 = (1<<PINF5)&PINF;    // Switch 2
                s3 = (1<<PINF6)&PINF;    // Switch 3

                // Switch depending on state
                switch (state) {
                        case DVR_STOPPED:
                                // TODO: Implement button/LED handling for record/playback/stop
                                PORTD &= ~(1<<PIND4);            // LED1 = OFF
                                PORTD &= ~(1<<PIND5);            // LED2 = OFF
                                PORTD |= (1<<PIND6);             // LED3 = ON

                                // TODO: Implement code to initiate recording
                                // Switch 2 pressed
                                if (!s2) {
                                        printf("Recording...");  // Output status to console
                                        dvr_record();                    // Initiate recording
                                        state = DVR_RECORDING;   // Transition to "recording" state
                                }

                                        // Switch 1 pressed
                                        if (!s1 && !play) {
                                                printf("Playing...");    // Output status to console
                                                dvr_playback();          // Initiate recording
                                                state = DVR_PLAYING;     // Transition to "playing"
state
                                        }

                                        play = 0;                        // Define play state
                                break;
                        case DVR_RECORDING:
                                // TODO: Implement stop functionality
                                // Switch 3 Pressed
                                if (!s3) {
                                        pageCount = 1;   // Finish recording last page
                                }

                                // Write samples to SD card when buffer page is full
                                if (newPage) {
                                        newPage = 0;     // Acknowledge new page flag
                                        wave_write(buffer_readPage(), 512);
                                } else if (stop) {
                                        // Stop is flagged when the last page has been recorded
                                        stop = 0;        // Acknowledge stop flag
                                        wave_write(buffer_readPage(), 512);      // Write final page
                                        wave_close();                            // Finalise WAVE
file
                                        printf("DONE!\n");                       // Print status to
console
                                        state = DVR_STOPPED;                     // Transition to
stopped state
                                }
                                break;
                        case DVR_PLAYING:
                                // TODO: Implement playback functionality
                                // Switch 3 Pressed
                                if (!s3) {
                                        stop = 0;                // Acknowledge stop flag
                                        printf("DONE!\n");       // Print status to console
                                        state = DVR_STOPPED;     // Transition to stopped state
```

```
                                }
                                if (!stop){
                                        play = 1;
                                        //printf("%d\n", OCR1B);
                                }
                                else {
                                        stop = 0;               // Acknowledge stop flag
                                        printf("DONE!\n");      // Print status to console
                                        state = DVR_STOPPED;    // Transition to stopped state
                                }
                                break;
                        default:
                                // Invalid state, return to valid idle state (stopped)
                                printf("ERROR: State machine in main entered invalid state!\n");
                                state = DVR_STOPPED;
                                break;

                } // END switch(state)

        } // END for(;;)

}
```

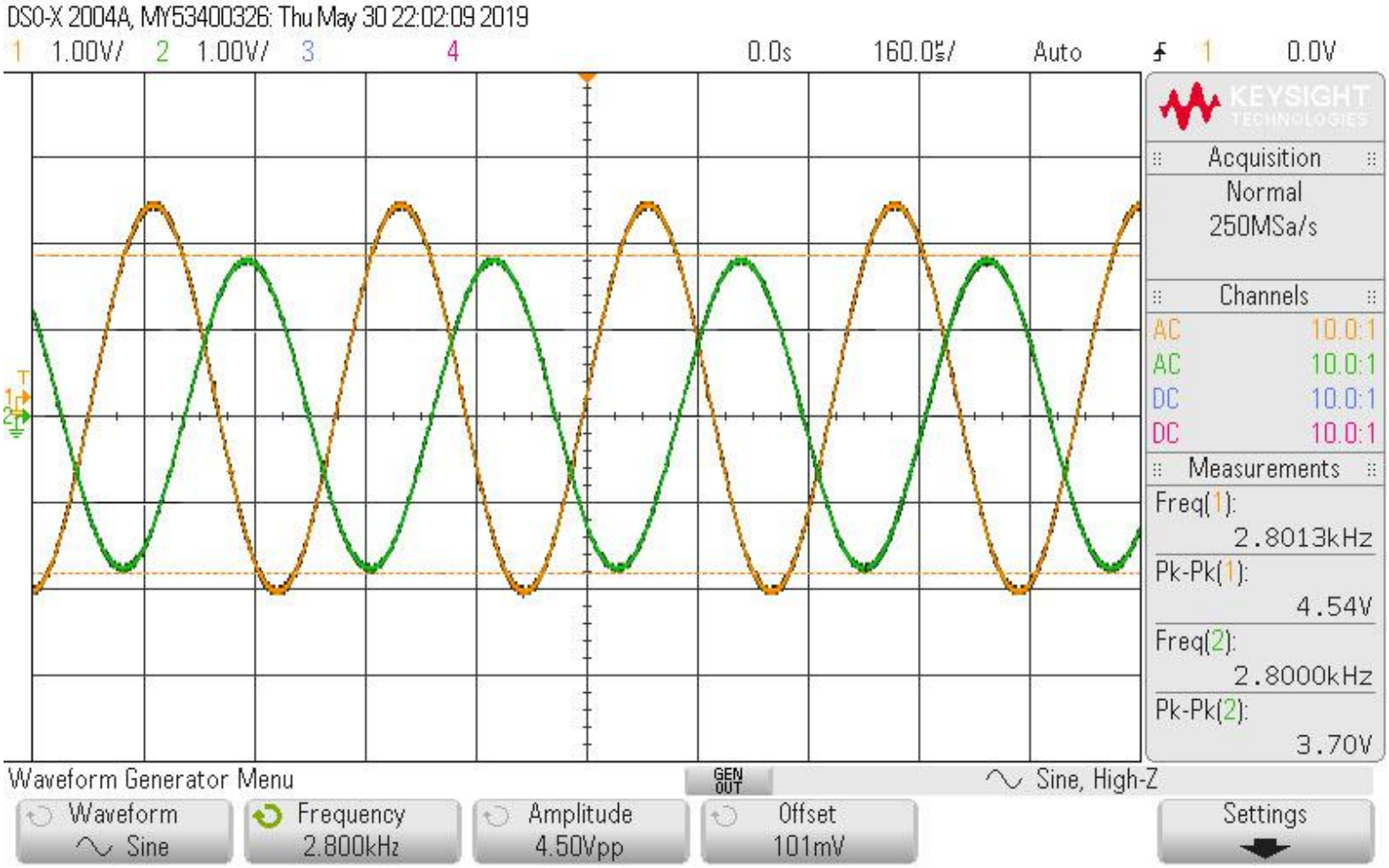
5 Filter Frequency Response

1.7kHz



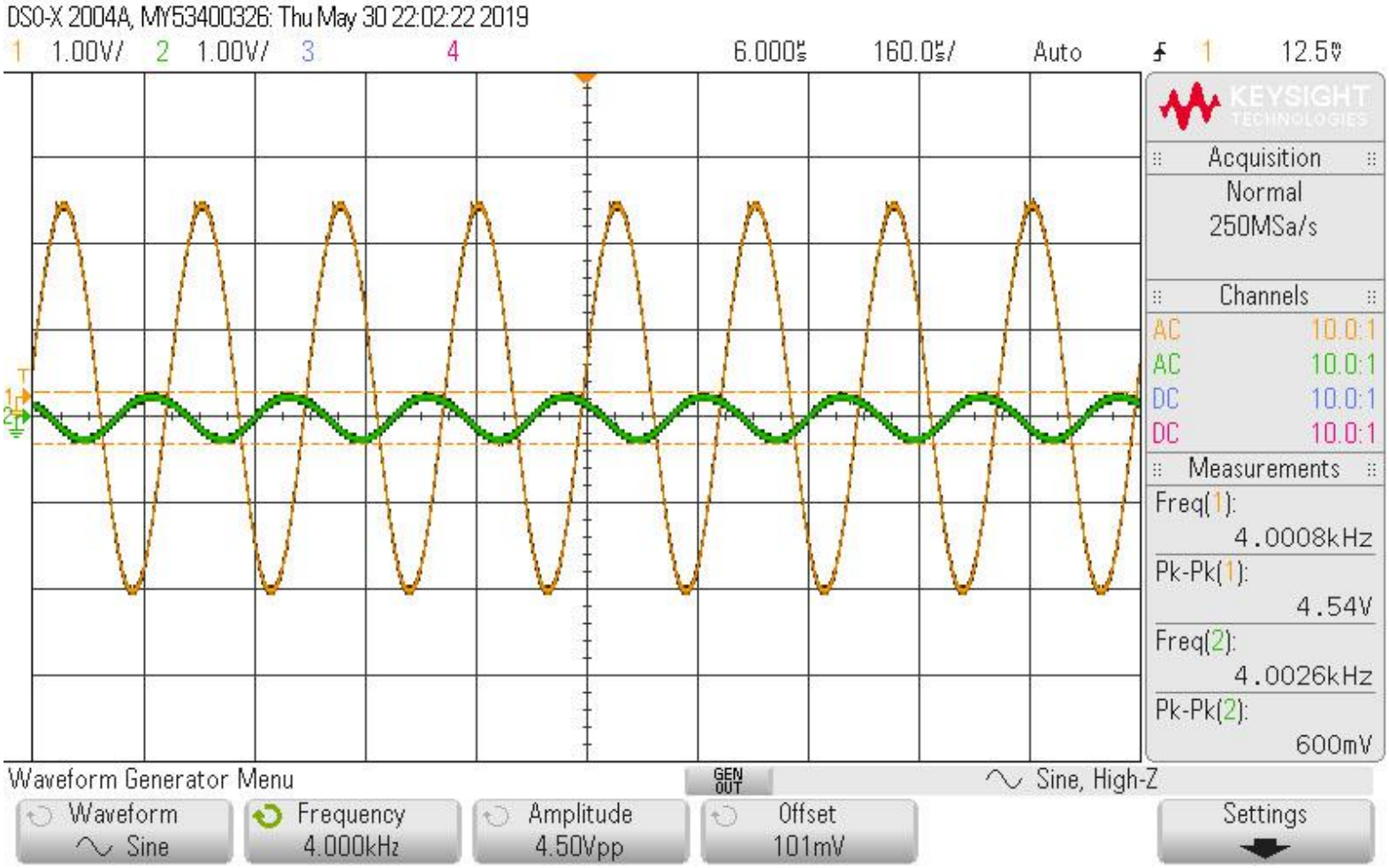
$$G(dB) = 20\log_{10}\left(\frac{4.57}{4.58}\right) = -0.02$$

2.8kHz



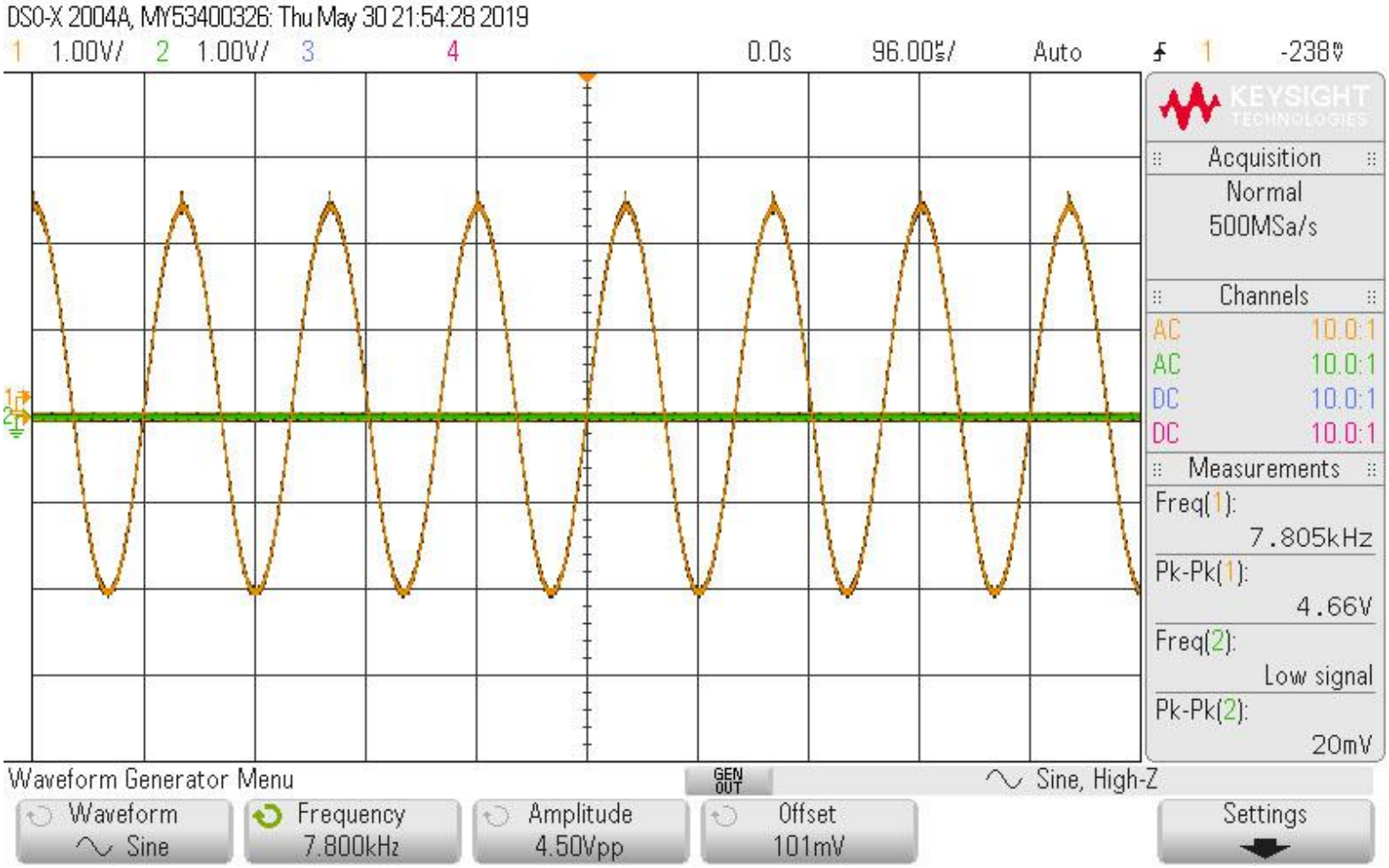
$$G(dB) = 20\log_{10}\left(\frac{3.7}{4.54}\right) = -1.78$$

4kHz



$$G(dB) = 20\log_{10}\left(\frac{0.6}{4.54}\right) = -17.58$$

7.8kHz



$$G(dB) = 20\log_{10}\left(\frac{0.02}{4.66}\right) = -47.23$$