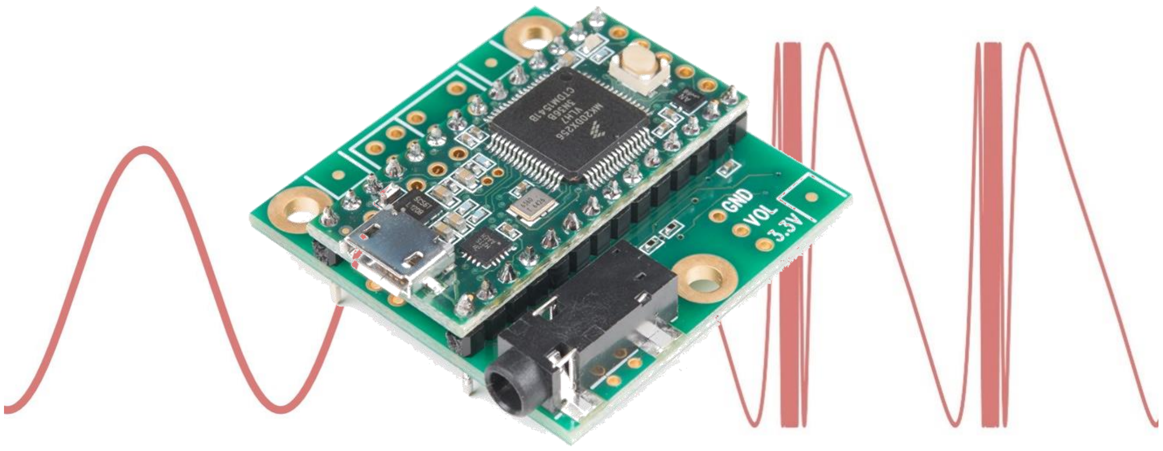


Teensy Dynamic Sound Effects Processor

Digital Signal Processing (DSP) with the Teensy 3.2 & Arduino Platform

Matthew Ooi

CMPEH 472 Final Project

12 December, 2017

Prepared for: Dr. Robert Gray

CMPEH 472; Pennsylvania State University, Harrisburg

Table of Contents:

Table of Figures:

1. Equipment Used

Qty	Equipment	Manufacturer	Model	Serial
1	Teensy 3.2 Microcontroller	PJRC	3.2	Unidentified
1	Teensy Audio Adaptor	PJRC	3.0 – 3.6	N/A
1	Tektronix MSO2012 Oscilloscope	Tektronix	MSO2012	C011261
1	DSO138 Oscilloscope	JYETech	DSO138	109-13800-00C
1	Waveform Generator	Keysight	33500B	E3620A
1	Midland Multimeter	Midland	23-105	49
1	(Micro-B) USB Cable	Anker	Generic	N/A
1	3.5mm Stereo Polytail	Generic	Generic	N/A
1	Stereo Headphones	Generic	Generic	N/A

Table 1-1: Equipment Used

2. Bill of Materials:

Qty	Equipment	Manufacturer	Model	Serial
1	Solderless Breadboard	Generic	Generic	N/A
10+	Jumper Wires	Generic	N/A	N/A
1	Joystick Sensor	Generic	N/A	N/A
1	128x32 OLED	Generic	SSD1306	N/A
1	DPDT Switch	Omron	Generic	N/A
1	4W Speaker	Anker	A7908	3TSP4Q
1	3.5mm Male Stereo Jack	Generic	Generic	N/A

Table 2-1: Bill of Materials

3. Problem Statement

Digital Signal Processing (DSP) is a method of applying digital processing to alter and/or obtain data from a signal waveform. Digital processing utilizes digital computation performed by computer processors.

DSP has many applications in the area of music production: in applying various algorithms to an analog audio waveform, transforming the pitch, tone, frequency, sample rate, and wave-shape of a signal to alter its perceived sound qualities.

The objective of this experiment can be summarized by three over-arching goals:

1.) Read-in an analog audio signal by converting analog information to digital data
2.) Apply transformation algorithms to the audio signal, audibly altering the signal qualities
3.) Output the modified audio signal on a speaker (analog device)

4. Abstract & Introduction

This experiment is intended to be a stepping stone into the basics of DSP; demonstrating various DSP concepts using the well-established Arduino Platform, with additional libraries and tools. As a stepping-stone into the principals of DSP, this experiment not only demonstrates real DSP concepts applied, but also serves as a hardware-foundation for more complex DSP effects that may be created using only software.

This experiment interfaces the Teensy hardware with a two-axis joystick, a 3-position selector switch, and a 128x32 monochrome OLED display to create a basic dynamic effects processor. A dynamic effects processor is a device often used in music production to modulate audio signals in real-time to produce different-sounding effects given some audio signal input.

The two effects (audio filters) created in this experiment are a "reverb" effect, and a "(guitar) distortion" effect. Additional effects can be created using the Teensy audio library.

This setup utilizes the Teensy 3.2 hardware: a microcontroller development board, comparable in function to OEM "Arduino" boards. The Teensy 3.2 surpasses every performance-comparable board in the current OEM-Arduino board family in terms of: physical size, speed, and cost. These three characteristics, along with a versatile audio library, and development-status stability make the Teensy 3.2 an ideal development board for this experiment. (Technical specifications of the Teensy 3.2 are discussed in section 5.2: Teensy 3.2 Platform).

The Teensy 3.2 is fully compatible with the OEM Arduino IDE using a software add-on called Teensyduino. In addition to the Teensy 3.2 microcontroller development board, this experiment uses the "Audio Shield" available for the Teensy hardware.

5. Investigation and Research

5.1. Objective

The objective and end-goal of this experiment is to produce a simple dynamic effects processor to modulate audio signals in real-time. This experiment will focus on creating two audio effects: A reverberation, or "Reverb" for short effect, and a "distortion" effect, similar to the sound of an electric guitar.

These dynamic effects will be applied to a digitized audio signal, originating from an analog audio source, using software executed on the Teensy 3.2 and Audio Shield hardware, and played back on an analog device: here, a powered 4W speaker.

5.2. The Teensy 3.2

The Teensy 3.2 is a microcontroller development board designed to operate within the existing Arduino platform. The Teensy family of microcontroller development boards offers functionality nearly identical to that of the OEM Arduino boards.

Figure 5-1: Teensy 3.2 Board [2]

The family of Teensy boards differs from most OEM Arduino boards in terms of physical footprint. The Teensy 3.2 is a fraction of the size, and cost of the Arduino DUE—the closest direct OEM equivalent in terms of qualitative technical specifications. Table 5-2 below lists the technical-specifications of the Teensy 3.2 board and also lists the fastest (in terms of processor clock-speed) OEM-equivalent board.

	Teensy 3.2	**Arduino DUE**
Price (per unit)	$19.80	$37.40
Processor	MK20DX256VLH7 Cortex-M4	AT91SAM3X8E
Operating Volt.	3.3 V	3.3 V
Speed	72-120 MHz	84 MHz
RAM	64 KB	96 KB
Flash Memory	256 KB	512 KB
# Digital I/O Pins	34	54
# Analog Input Pins	21	12
Length	35.56 mm	101.52 mm
Width	17.78 mm	53.30 mm

Table 5-1: Teensy 3.2 Technical Specifications versus Arduino DUE

5.2.1. Teensy Software

The Teensy family of boards is fully compatible with the Arduino IDE. However, the Arduino IDE requires an add-on called Teensyduino: maintained and distributed by PJRC [1]. Teensyduino allows the OEM IDE to work with the family of Cortex processors used to drive the Teensy boards. The Teensyduino add-on is available through the PJRC website (www.PJRC.com) [1].

5.2.1.1. Teensy Audio Library

The Teensy microcontroller family also features an exclusive, graphical audio library designed for use with the Audio Shield accessory. The audio library is different from other Arduino libraries in that it features an online drag-and-drop GUI (Graphical User Interface) with the ability to export Arduino code based on the graphical flow diagram.

The Arduino code generated by this online library is a collection of unique functions (called in the IDE as classes) declared at the start of the Arduino sketch. This collection of classes defines the flow of sound through the system.

This library provides the framework to create very advanced signal processing systems, able to be executed on the Teensy hardware platform. The library consists of ten categories:

Category	Description
Input	Functions to control audio input (source)
Output	Functions to control audio output
Mixer	Combines up to four audio signals, each with adjustable gain/attenuation
Play	Play audio files stored on different locations of expandable flash memory
Record	Collect audio data to send to Arduino
Synth	Simulate variety of sounds; frequencies, drums, strings, or custom waveform
Effect	Variety of pre-packaged audio effects
Filter	List of adjustable signal filters
Analyze	Monitor signal properties, compute Fast Fourier Transform (FFT)
Control	Control various, similar audio IC chips

Table 5-2: Teensy Audio Library Function Categories

Within the body of the Arduino Sketch, the audio library functions may be called to modify a function parameter, or return a value pertaining to a signal quality. For instance, the function “Peak” returns the highest peak-to peak amplitude since the last time the signal was read. [18]

5.2.2. Teensy 3.2 Hardware

In terms of functionality, the Teensy 3.2 is nearly identical to other boards in the OEM Arduino family. However, while most Arduino OEM boards operate on a standard of 0V—5V digital logic, the Teensy 3.2 digital I/O pins operate on a standard of 0V—3.3V.

The 3.3V digital logic standard is based off newer CMOS technology that operates with lower power, and at a lower voltage than traditional 5V TTL logic. 5V TTL digital logic regards a range of 2V—5V as a digital 1 or HIGH, and a range of 0V—0.8V as a digital 0 or LOW. 3.3V logic on the other hand, regards 2V—3.3V as a digital 1 or HIGH, maintaining a range of 0V—0.8V to represent a digital 0 or LOW. [3] While not all 3.3V devices are 5V compatible, the Teensy 3.2 is 5V-tolerant on the digital I/O pins; accepting 5V signals without damage to the development board. [4] Additionally, like other Arduino OEM boards, the Teensy board is powered from a standard USB.

The Teensy 3.2 utilizes a Cortex M4 processor produced by Arm. The Cortex M4 is high-performance, low-power, 32-bit processor, featuring a DSP instructions (to optimize DSP operations). [5]

The default operating speed for the Teensy 3.2 is 72 MHz. After installation of Teensyduino, several menu options unique to the Teensy hardware are made available through the menus in the Arduino IDE. Through the tools menu, the user may change the clock speed of the (Teensy) processor. There are several choices of clock speeds, ranging from 2 MHz to 120 MHz. The available 96, and 120 MHz choices are regarded as "overclocking" the processor: that is, running the processor at speeds faster than the manufacturer rating.

The creator and developer of the Teensy platform, Paul Stroffregen, made the overclocked-options available to the user only after testing that these clock speeds did not harm the processor or incur any significant additional heat. [6]

Processor clock speed is a vital specification for DSP applications. Clock speed is representative how many iterations of an operation a system can execute in one time period. Here, Megahertz is an integer multiple of a million cycles per second. Therefore, when operating at 120 MHz, the Teensy is theoretically capable of 120**E**6 operations per second.

For DSP systems to give the user the impression of operating in "real time" the systems must perform the system's computational operations with a latency of less than approximately 10ms.

5.3. Digital Signal Processing (DSP) Basics

Digital Signal Processing (DSP) is means by which a signal waveform is digitally manipulated using a computer; applying a function or algorithm to the waveform to enhance, diminish, or measure certain qualities of the original signal; thus acting a filter.
These qualities are the signal:

1.) Period
2.) Frequency
3.) Amplitude

(Real-time) signal processing, specifically used in music production is not a new technology. Analog signal processing has existed for many years commonly in the forms of guitar "effect" pedals, synthesizers, and voice-modulators.

DSP, on the other hand is a growing technology as faster, and more advance systems emerge. Low-latency DSP requires the ability to process signals quickly using either high-performance CPUs, or low-overhead CPUs like those found on microcontrollers.

The chart below shows the system stages of a standard DSP system:

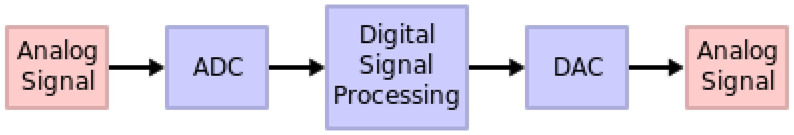


Figure 5-2: DSP System Stages

5.3.1. Analog to Digital Conversion (ADC)

Commonplace or "daily-interaction" waveforms typically originate from analog sources: musical instruments, mechanical vibrations, light from light bulbs etc. Before such analog waveforms can be altered, they must be converted to a stream of digital information. Thus, DSP employs existing analog-to-digital conversion technology.

Analog signals contain an infinite number of data points, each data point representing an absolute value in the time domain. Analog audio signals contain continuous variables, in the change in voltage compared to time. Each point on an analog audio waveform exactly represents a sound as it is heard through a speaker.

Given that analog signals are scalar quantities, they are susceptible to transmission degradation and interference.

One method to circumvent this issue, analog signals may be represented as digital information using an Analog to Digital Converter (ADC). The resolution, or "accuracy" of the converted signal depends on the maximum resolution of the ADC. For instance, an 8-bit ADC can only represent an analog signal with up to 8-bits of resolution or 256 unique values. Whereas, a 16-bit ADC can represent an analog signal with up to 16-bits of accuracy or resolution or 65,536 unique values. [8]

5.3.2. Digital to Analog (DAC)

Most DSP applications apply the digitally-processed waveform to some type of analog device, or transmit the information to another device or system for monitoring and/or measuring. Applications requiring high-fidelity signals typically employ high-resolution digital to analog converters (DACs).

The basic functionality of a DAC is to take binary (digital) data, represented in 0's and 1's and convert it to a smooth, analog waveform. DACs perform a signal operation "opposite" to that of Analog-to-digital converters. In a DSP system, a DAC resolution is limited by the ADC. Therefore, it is important to ensure that a DSP system has an ADC that is at least greater than or equal to the bit-resolution of the DACs.

5.3.3. System Latency

One problem that many DSP systems face is minimizing system signal-latency to an acceptable level.

Many electronics systems, from house lights to moving images on a TV screen appear to occur in “real time.” That is, the action is perceived to take place at the very moment the switch is flipped “ON” or the channel on the television is changed. However, all systems contain a degree of latency measured in time.

The biology of the human brain has a minimum perceptive index. In other words, below a certain threshold, humans cannot perceive any latency and thus actions below this threshold appear to take place “instantaneously” or, in “real time”.

Human perception of audio latency varies slightly from person to person. Therefore, there is no absolute minimum threshold for human audio-latency perception. However, based on a 2015 article by Thomas Burger [12] VoIP (Voice over IP) providers will not accept a call latency greater than 150ms. Moreover, according a professional audio-production company, PreSonus, noticeable audio signal delay begins around the threshold of 10-13ms for most individuals. [13]

5.4. Teensy Audio Shield

The Teensy Audio shield is a small circuit board adding high-quality 16-bit, 44.1 kHz sample rate (CD quality) audio to the Teensy hardware family. [7] The Audio Shield has a microSD card slot to store audio files, a 3.5mm headphone jack for audio output, and through-hole solder points to add a: microphone, stereo line-level input, and stereo line-level output.

Line-level audio differs from amplified-audio in that: line-level audio is an unamplified signal, where amplified-audio is suitable to directly power speakers.
Line-level audio contains all the information necessary to represent a sound, but is at a potential (voltage) and/or current too low to drive a speaker.

5.4.1. SGTL5000 Stereo Codec Chip

The Teensy Audio Shield is driven by the SGTL5000 Low Power Stereo Codec with Headphone Amp integrated circuit. This chip is available in an SMD package, and already assembled with the fully-functional out-of-the-box audio shield.

This stereo codec chip contains an on-board ADC, and DAC capable of providing 16-bits of audio signal resolution. The SGTL5000 also features an on-board programmable mic-gain input, a stereo headphone amplifier (line-level output), and an I^2S (digital I/O) port to communicate with other digital audio chips. [10] Section 5.4.2 below expounds on the I^2S protocol. To see the full technical specifications of the SGTL5000 IC, one may refer to the datasheet included under References. [10]

5.4.2. I^2S Bus

This section provides a very brief introduction to the I^2S (Inter-IC-Sound) bus.

The I^2S bus protocol is similar in function to the I^2C bus: it transmits and receives digital information between digital devices. [11] The I^2S bus differs from the I^2C bus in that it contains only audio signal information. Control and telemetry data is reserved for use on the I^2C bus.

Like I^2C, the I^2S protocol is a standardized, and widely accepted digital transmission protocol for use with audio systems. Moreover, I^2S implements the same control scheme as I^2C: a system assigns a bus master, commanding other I^2S slaves.

The I^2S protocol only requires three wires:

1.) Clock (SCK)
2.) Word Select (WS)
3.) Serial Data (SD)

5.4.2.1. Clock (SCK)

The clock signal for a "Slave" device is an external signal, commanded by the "Master". Thus, any Slave device on the bus is capable of assuming the role of the Master given a specific clock pulse. [11]

5.4.2.2. Word Select (WS)

The Word Select line for the I^2S bus indicates which channel of stereo audio is being transmitted. [11]

The I^2S protocol uses the convention below to differentiate between left-channel, and right-channel audio:

- (WS) = 0 (left)
- (WS) = 1 (right)

5.4.2.3. Serial Data (SD)

The Serial Data line on the I^2S bus transmits the digital signal on either the leading edge, or falling edge of the of the clock signal.

This data is transmitted using two's compliment, sending the MSB (Most Significant Bit) first. This convention accounts for different word lengths between the system master and receiver. When a transmitted signal word length is less than that of the system, the missing data is represented in the system LSB (Least Significant Bits) as zeroes. [11]

5.5. Audio Signal Qualities

5.5.1. Clipping

Signal processing typically requires manipulating a signal waveform. For the applications in this experiment, audio signals will be modulated using various transformation functions to alter the audio signal characteristics.

In transforming, or multiplying a signal by a gain, it is necessary to monitor the signal amplitude such that the signal maximum does not exceed the systems limits. When a signal's amplitude surpasses the limits of a system, the signal will undergo what is called *clipping*. In most applications, clipping is the byproduct of over-amplification (too much signal gain) or constructive interference, producing waveforms with maximum amplitude outside of the systems limits.

When a signal clips, any waveform information above the system maxima is lost, and the signal saturates at the "ceiling" of the system limits. Clipping produces a corner in the signal, and in the case of audio signals, produces a harsh distorted sound as the loss of signal information becomes audible.

In this experiment, the system limits are defined on a scale from zero to one hundred; or, 0—100% gain. It is good-practice to remain well below the maximum gain (one) to ensure no clipping occurs, even in the instance of constructive interference.

5.5.2. Attenuation

To combat signal clipping, it is necessary to *attenuate* the audio signals. Signal attenuation is the action used in reference to reducing the amplitude of a signal.

Attenuation is used to control the constructive interference between waves the occurs in a multi-stage amplifier system such as the one used in this experiment. By attenuating the signal at each stage of the amplification process, one can ensure that the signal integrity is maintained at every point in the system, and no clipping will occur.

5.5.3. Multistage Audio-Mixer System

Many amplifiers consist of more than one signal amplification stage. Hardware audio amplifiers are analog devices used to control the amplitude of a signal. Amplifiers control the amplitude of a signal by allowing adjustment of the system's gain.

In this experiment, the audio amplifier is represented by a "Mixer" function block. The Mixer function block behaves like a summing amplifier with adjustable gain. The gain in the system is measured on a scale of 0-1.0, or 0—100% of the maximum voltage for the system on-board amplification chip. The maximum measured output voltage for this system is measured to be 1.2V.

Therefore, in order to avoid clipping, the signal must remain below the maximum system output of 1.0 at every stage of mixing. In terms of value: when the gain of each channel in a mixer is added, the total sum must be less than or equal to 1.0.

Each mixer object has four input channels, and one output. Each input channel is numbered from 0 to 3. Each input terminal of a mixer can only receive one signal. However, the output of a mixer can have unlimited signal connections.

The Teensy audio library mixer function permits values of ±32,7670.0 for the gain. Any value above 1.0 amplifies the signal. Any values between 0 and 1.0 attenuates the signal. This experiment does not require signal amplification as both the input and output are line-level audio signals. Amplifying a line-level input signal above 1.0 may exceed maximum the line-level output voltage achievable the Teensy.

5.6. Sound Reverberation

5.6.1. Naturally Occurring Reverberation

Natural sound reverberation occurs when there is discrepancy delay between sound waves heard by a point receiver. A point receiver may be either a device such as a microphone, or a human standing at a certain point; both point receivers interpret sound waves from a source, or sources at that specific location.

Reverberation, or "Reverb" for short is the result of sound waves reflected off objects in the surrounding area: walls, floors, buildings, cars, etc. Hard, planar, objects are ideal for reflecting sound waves.

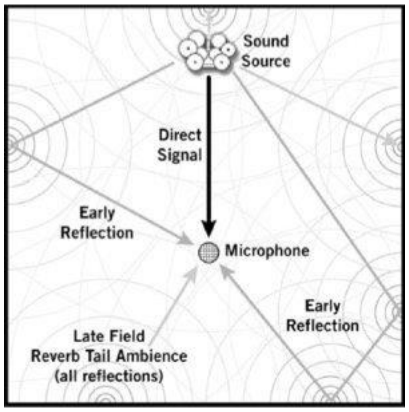


Figure 5-3: Naturally Occurring Reverberation

Reverb is often used in reference to several sound waves constructively, and deconstructivity interacting with each other, as well as a primary source to create a rich "three dimensional" sound effect. Thus, true reverb is purely random as there is an infinite number of sound waves, each reflected at different angles, and traveling at different speeds based on the reflective surface.

While "true" reverb cannot be intrinsically recreated, it is possible to simulate reverb using hardware and software. For instance, reverberation guitar-effect pedals have existed for many years, using analog circuitry to re-create an adjustable delay line similar to that of a natural reverb.

5.6.2. Artificial Reverb Audio Effect Using Software

Software-simulated reverb effects are available in many audio-production software suites. While these reverb effects are often of high-fidelity, most are designed for post-production application and require operation within a computer operating system environment rather than a stand-alone hardware package.

There exists several, simple, open-source reverb filter-effect designs. Of these is one design titled "Freeverb" available through Stanford's website [14]. Freeverb is a filter design based on Schroeder artificial Reverberator designs, pioneered by Manfred Schroeder and Ben Logan in the 1960's.

This experiment does not use the Freeverb effect as it is documented. This experiment models several design aspects of the Freeverb filter design, but does not attempt to copy the filter design.

5.6.3. Hardware Limitations of Artificial Reverb

Freeverb is considered a very "basic" reverberation effect in comparison to professional, non-open-source, alternatives. In early experimentation, it was found that even the original Freeverb filter design vastly exceeds the memory capabilities of the Teensy hardware platform.

Though advanced computational reverb effects govern several continuous variables, they are not primarily processor-power limited. Rather, advanced reverb effects demand a large amount of RAM to cache of enormous number of delay tap lines, and each line's late-reflections. [15]

5.6.4. Constructing a Simple Reverb

In order to simulate a reverb effect, it is necessary to design and audio loop containing several delay-tap lines. In simulating a reverb effect, each delay-tap line represents a distinct echo (sound wave pulse) returning to the ear (or detector) of the observer. Therefore, the more delay-tap lines included in the reverb effect, the more distinct echoes occur in the final audio mix.

Moreover, in order to simulate a reverb effect using software it is necessary to incorporate a feedback loop. The purpose of a feedback loop is to simulate a continuous, decaying, "copy" of the original source sound to the ear of the observer. Natural reverberation does not only receive a single delayed sound pulse from the point of origin, but also detects second, third, and even fourth, reflected sound waves fully decay and cease to propagate.

Thus, not only is it necessary to include a feedback loop, but it is vital to attenuate the feedback gain to avoid chaotic, infinite, constructive interference. Without attenuation, the feedback signal exponentially multiplies with the original signal with each iteration through the feedback loop.

Lastly, in order to simulate a reverberation effect, each delay line must "hold onto" each copy of the source signal for a time that is *not* an integer multiple of another delay. Each delay must delay the source signal by a continuous, irrational, value to avoid any possibility of deconstructive interference. [15]

While natural reverberation deconstructivity interacts with other sound waves in the environment, artificial reverb filters lack the advantage of: an infinite wavenumber, truly random wave propagation, frequency modulation based on the environment. Selecting continuous, irrational, delay line values gives the final reverberated signal the appearance of being random, and prevents the limited number of delay tap lines from becoming integer multiples of one another based on the current feedback-line iteration.

Advanced artificial reverb filters that incorporate the aforementioned variables exist in many audio-engineering software suites. However, variable-control of wavenumber, frequency modulation, and advanced wave speed randomization exceeds the scope of this experiment.

5.7. Distortion Audio Effect

A "distortion" audio effect is the name commonly associated with the iconic steel-grunge sound emanating from an electric guitar. This iconic sound is in practice is little more than a wave-clipping effect. [16] This experiment employs a filter similar to that of a guitar distortion pedal; here, called a "bitcrusher" effect in the Teensy audio library.

There are two types of signal clipping used in a guitar distortion effect (also similar to an "Overdrive" effect): clipping, and soft clipping.

5.7.1. Clipping

Intentionally clipping a signal to produce a distortion effect is called clipping, and produced a waveform with sharp corners at the maxima and minima of the wave. The ceiling of the clip is determined by the limiting voltage, or the strength of the distortion effect. [17]

5.7.2. Soft Clipping

Soft clipping is very similar to "regular" clipping in that the signal saturates at the ceiling of the clip, determined by the strength of the effect. However, unlike "regular" clipping, *soft* clipping eliminates the sharp corners produced by suturing the signal, providing a degree of antialiasing to the waveform. [17] Soft-clipping produces a signal similar to that of a "regular" clip, but with softer tonal qualities.

5.7.3. Teensy Audio Library: Bitcrusher Effect

The Bitcrusher effect is a predefined function in the Teensy audio library. The filter effect modulates a waveform by controlling the number of digital bits to "crush" (to set to zero), and sample rate at which the audio signal is returned to the system.

By zeroing bits from the original signal, and lowering the sample rate, the audio signal becomes distorted with sharp(er) corners, and clipped maxima and minima.

Lowering the sample rate of the signal vastly reduces signal resolution, also producing a distorted sounding signal.

5.8. Logarithmic Gain

Sound waves travel at speed proportional to the medium in which they propagate. Humans detect, or hear these sound waves on a logarithmic scale. For instance, sounds twice as large in magnitude, are not perceived by the human ear as *twice* as loud. [19] This convention of the human hearing allows people to hear very quiet sounds, and tolerate very loud sounds like those at a rock concert.

This experiment utilizes logarithmic hearing-perception to bridge the gap between linear effect-adjustment (input), and perceived effect sound quality. The joystick sensor used in this experiment is simply a two-axis potentiometer with linear adjustment.

Adjusting the effect parameters on a logarithmic curve allows the user to perceive a great change in effect for a small linear movement. Therefore, the distortion effect is biased towards fewer bits crushed for initial joystick movement, and greater number of bits crushed per degree of movement at the far end of the scale.

6. Construction and Implementation

6.1. Defining the Scope

The scope for this experiment focuses on using the Teensy hardware platform in conjunction with the Teensy Audio Shield, and audio library to create a simple DSP system; modulating line-level (analog) audio in real-time to create a reverb effect, and a distortion effect, displaying the current system data on a 128x32 OLED display.

This experiment focuses on the characteristics of audio signals, waveforms, and signal processing at an entry-level project for those with little to no prior experience with DSP. Yet, designing a system flexible enough for future, advanced development with more complex processing features and algorithms.

The construction of this system implements, but does not focus on the properties of OLED displays. Only basic functionality will be discussed in this section.

In summary, the goals for this experiment are defined with the following statements:

1.) Construct a DSP system capable of transforming analog audio signals in real time
2.) Design and implement two effects (and/or “filters”): a reverberation, and distortion effect
3.) Implement a Human-Interface Device (HID) to control the sound-effect parameters
4.) Display relevant system data on a 128x32 OLED display
5.) Output the modulated audio signal to an analog device for listening over a speaker

6.2. System Design

The logical design for flow of data in this DSP system consists of two layers:

1.) Audio System Flow
2.) Program System Flow

6.2.1. Audio System Flow

The first layer in this system is the Audio System Flow. The Audio System flow controls the behavior of a signal in how it progresses through several stages of modification and amplification/attenuation.

The audio system flow is designed using the Teensy audio library (abbreviated TAL for this documentation) GUI.

Microcontrollers are finite state machines, designed to execute one task at any given time. Therefore, to design a multi-effect audio system flow for a microcontroller, the audio system must be analyzed as a single, complete system, contained in one layout. While this is suitable for small-scale systems such as this experiment, it is inefficient for larger, multi-effect systems as every effect must be computed simultaneously.

No parameters for any function need to be defined using the TAL GUI. The purpose of this tool is layout flow diagrams for entire systems. System parameters are defined within an Arduino Sketch.

Below is the flow diagram for the audio system used in this experiment:

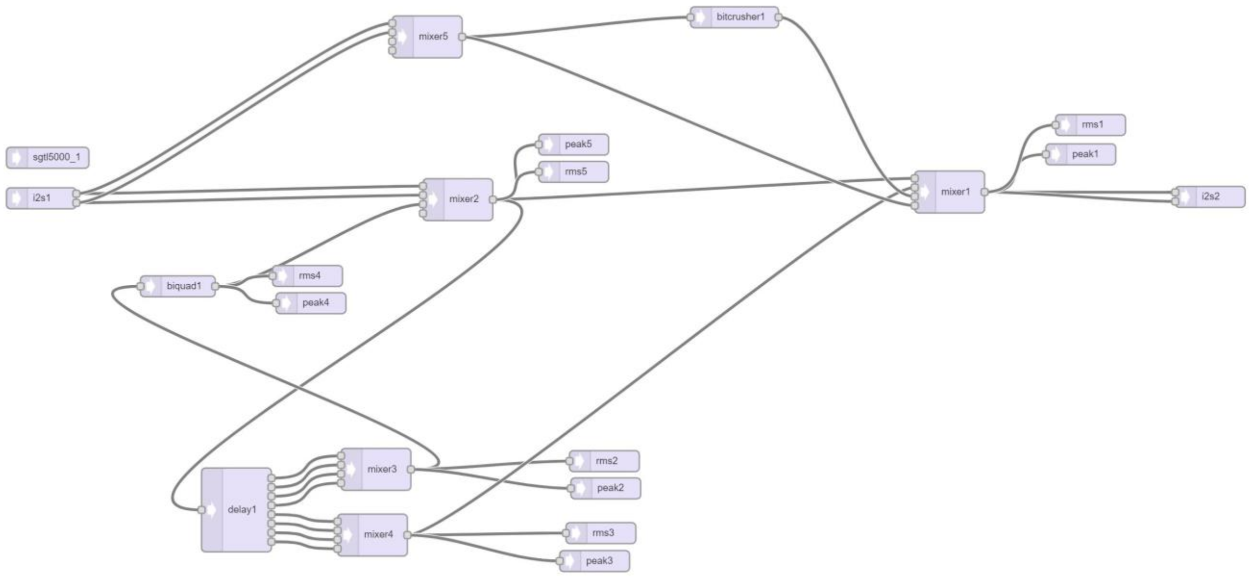


Figure 6-1: Audio System Flow Design

Below is a labeled copy of the audio system flow, defining each stage of the system:

Note, The objects “Peak” and “RMS” are objects used for signal measurement. Signal measurement is an important part of this system so that signal flow may be “tuned” to the desired behavior.

The object “Peak” is used to measure the peak amplitude in a signal. The “RMS” object is used to measure the *root mean square* (RMS), or “effective” signal magnitude.

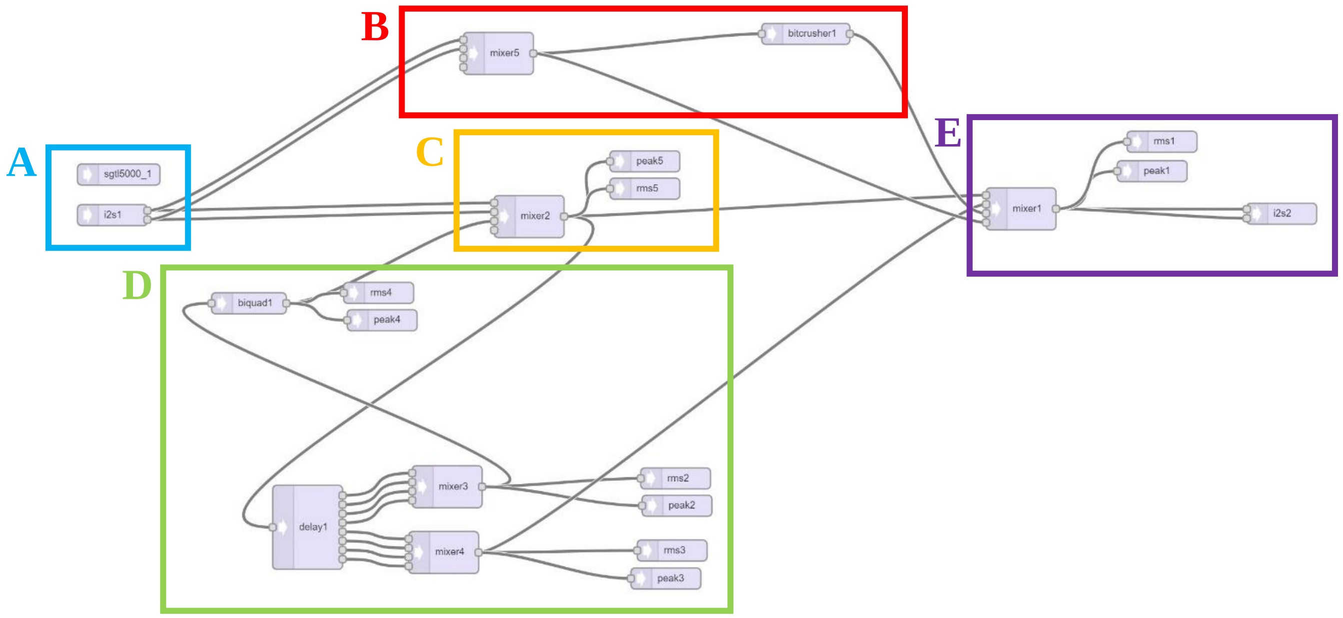


Figure 6-2: Labeled Audio System Flow Design

Label	Audio System Stage
A	Input
B	Distortion Effect
C	Pass-Through
D	Reverb Effect
E	Output

Table 6-1: Sound System Flow Label Definitions

A. *Input Stage*

The first step in designing any audio-flow using the Teensy audio library (TAL) is to include the singular control block “**sgtl5000**”. Including this block requests permission to take manual control of the SGTL5000 IC on the audio shield. No other objects in the TAL interact with the **sgtl5000** object.

Next, an input source for the audio signal must be defined. The TAL has several choices for input choices, here the digital audio protocol **I^{2}S** will be used for the input source. The I^{2}S input defers to the “LINE IN” (L/R) ports on the Teensy audio board.

This experiment negates individual channels for left and right audio channels, bridging and combining these channels into a mono-audio signal.

Note: Each node on the input or output of an effect in the TAL represents a single channel. Take for instance, the **i^{2}S** input object with two output nodes: one node represents *left* channel audio input, and the other represents *right* channel audio. The node definitions are listed under "**Audio Connections**" for each object in the TAL.

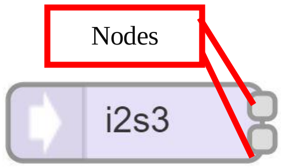


Figure 6-3: Teensy Audio Library Object

B. Distortion Stage

The distortion stage of the system first bridges the left and right audio signals using **mixer5**. The entirety of the distortion effect is contained in the (predefined) **bitcrusher** effect. Here, bitcrusher is defined by **bitcrusher1**.

C. Pass-Through Stage

The pass-through stage of this system is responsible for only bridging the left and right channels of the input. Following signal-bridging is the output stage.

D. Reverb Stage

The reverb stage of the system consists of three primary objects:

1.) Delay
2.) Mixer
3.) Biquad (Lowpass) filter

The 8-channel delay-tap object is the primary driver behind the reverb effect. In this system design, the first four channels, each representing a unique duration delay, are passed back to the first-stage mixer as feedback. These first four channels are "aggressively" attenuated (by values less than 0.5) to prevent exponential constructive interference.

The feedback is then summed with the original signal, and re-enters the delay for a second time. This process continues indefinitely until a given signal completely decays.

The feedback loop passed through a Biquad filter, configured as a low pass filter. This filter is ideal for limited-power finite state machines because it has only a single summation point. [21] Thus, filtering has marginal impact on system latency and performance.

A reverb feedback, low pass filter, is objectively optional. However, early testing revealed problems with low-frequency feedback signal decay. Low, bass frequencies took an extensive amount of time to fully decay, and constructively-interfered with incoming signals, ultimately producing unwanted clipping.

The second four channels of delay also represent unique duration delays, and are passed directly to the output-stage mixer, and summed with the original signal.

The TAL currently offers a "Reverb" object that may be used to apply a synthetic reverb to any incoming audio line signal. However, this effect requires more RAM (exact value unknown) than the 64 KB RAM on the Teensy 3.2. Therefore, this effect is currently incompatible with the Teensy 3.2. Though, it may be used on the high-end Teensy boards with more RAM (such as the 3.5 or 3.6).

E. Output Stage

The output stage amplifier sums the signals from the distortion, pass-through, and reverb stages. The output mixer, **mixer1** sends the signal data to an I^2S object. The I^2S object passes through the SGTL5000 internal DAC, sending line-level audio to the LINE OUT (L/R) and pre-amplified headphone jack on the Teensy Audio Board.

In order to differentiate each effect, **mixer1** also called the final-stage mixer in this experiment, uses gain control to select which channels to pass to the speaker output. For instance, when the “Reverb” effect is currently active, the gain of channel 2, form the Distortion effect, is set to zero.

Likewise, when the distortion effect is selected, all channels except 2 and 3 are set to a gain of zero.

6.2.2. System Layout

The final system when assembled on a breadboard is shown in the figure below:

Note: The neon-green colored box indicates an audio amplifier circuit that is in no-way electrically connected to the audio system documented in this experiment. It is an audio amplifier circuit undergoing testing for further system expansion.

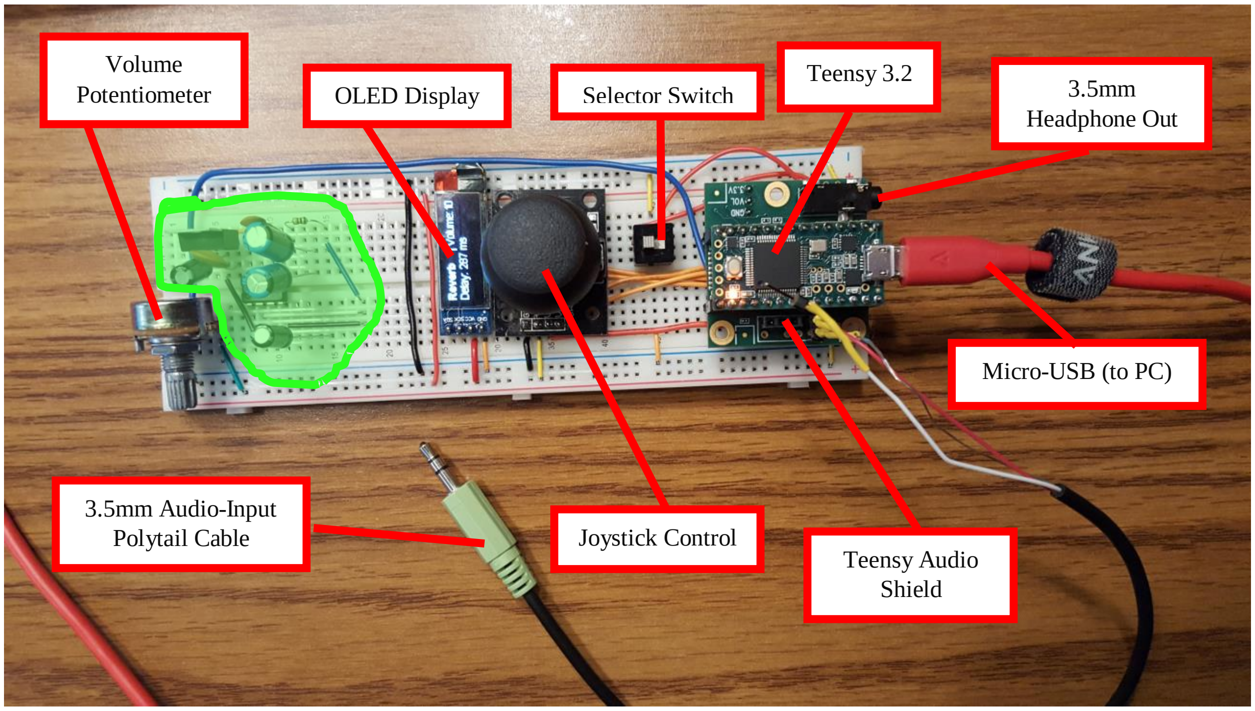


Figure 6-4: Audio System Setup

6.3. Circuit Design

The circuit for this experiment is made up of six primary components:

1.) Teensy 3.2 Microcontroller Board
2.) Teensy Audio Shield
3.) 3-position selector switch
4.) 2-axis joystick
5.) 10K Potentiometer
6.) 128x32 OELD display

The Teensy 3.2 and audio board are responsible for driving all logical the operations. There are three human-interface devices (HID): A joystick, potentiometer, and selector switch.

The 2-axis joystick functions as a variable resistor to manipulate the audio effect parameters. The joystick also features a momentary pushbutton switch when the joystick "hat" is depressed directly down. This momentary switch serves as a "MUTE" toggle switch.

The 10K (linear) potentiometer functions as a volume-control knob.

The selector switch functions as a switch to choose between: Reverb (top position), Pass-though (middle position), and Distortion (bottom position).

The Teensy 3.2 features on-board pullup resistors, definable in the Arduino Sketch code. Therefore, external pull-up resistors for the switches are unnecessary.

All features in this experiment are powered directly from the Teensy 3.2, via USB.

Shown on the following page is the schematic for this circuit.

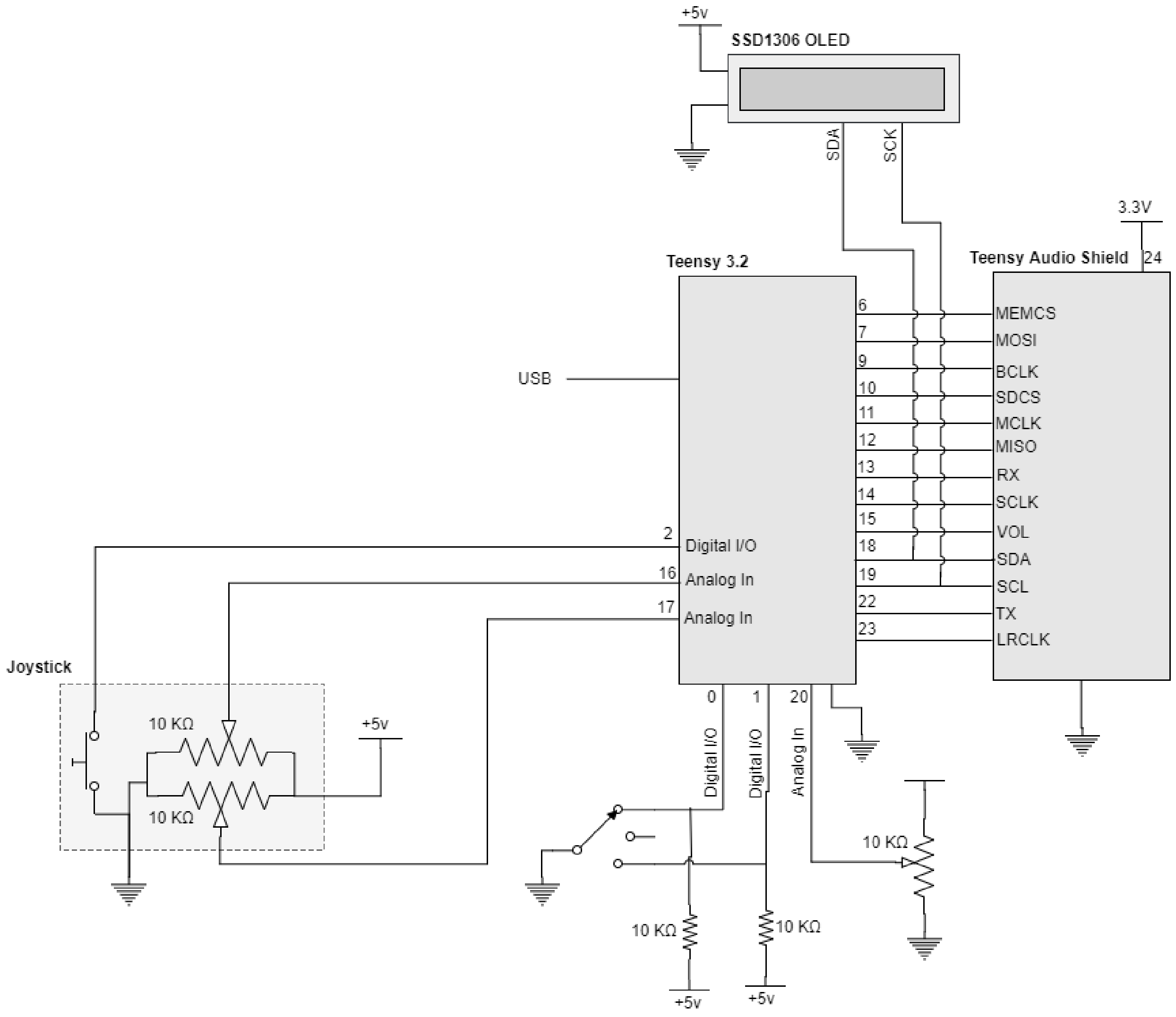
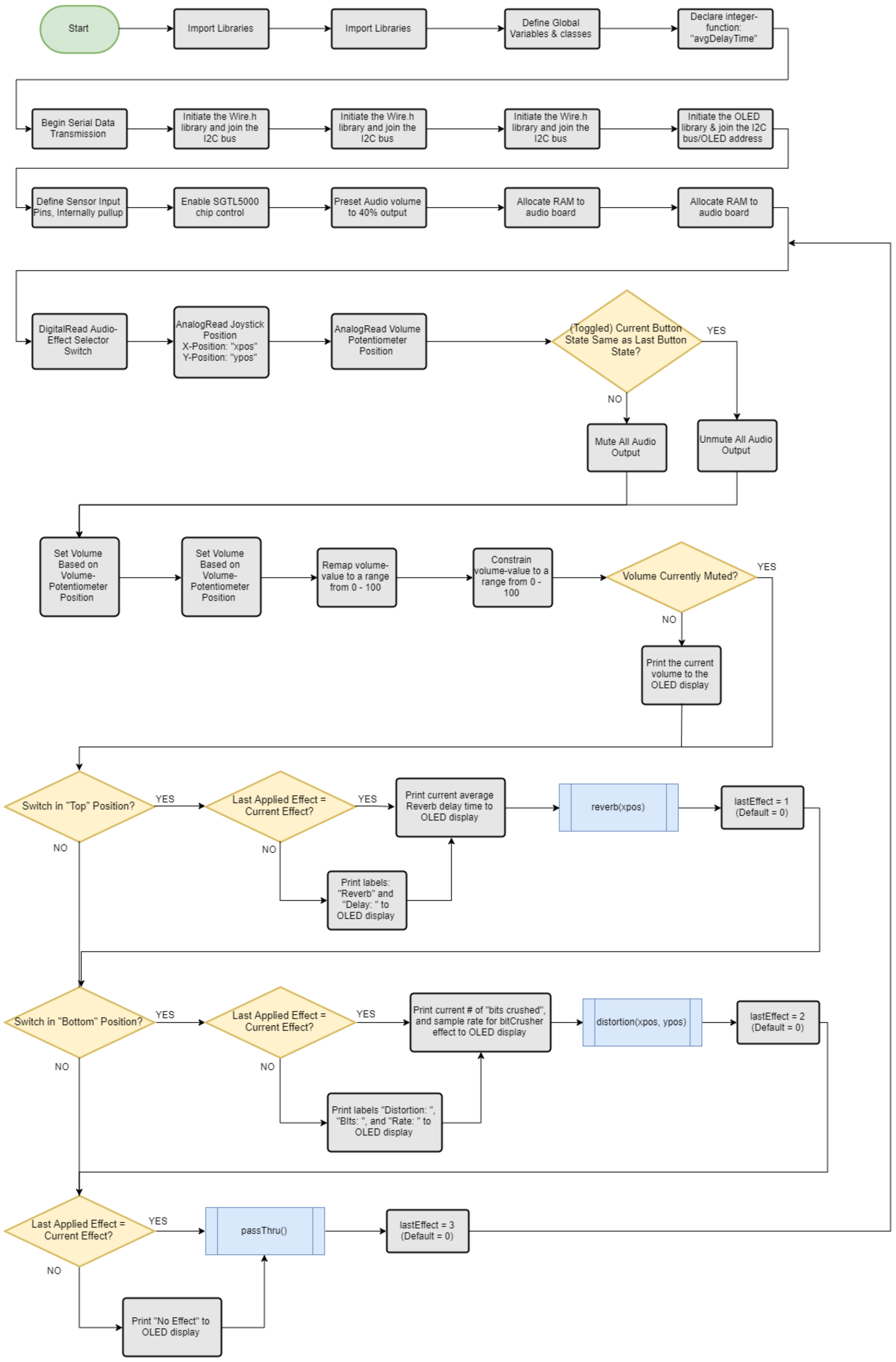


Figure 6-5: Circuit Schematic

6.4. Arduino Code

The Arduino code for this system is extensive and can be best represented with the following flow chart (next page):



Start
Import Libraries
Import Libraries
Define Global Variables & classes
Declare integer-function: "avgDelayTime"
Begin Serial Data Transmission
Initiate the Wire.h library and join the I2C bus
Initiate the Wire.h library and join the I2C bus
Initiate the Wire.h library and join the I2C bus
Initiate the OLED library & join the I2C bus/OLED address
Define Sensor Input Pins, Internally pullup
Enable SGTL5000 chip control
Preset Audio volume to 40% output
Allocate RAM to audio board
Allocate RAM to audio board
DigitalRead Audio-Effect Selector Switch
AnalogRead Joystick Position X-Position: "xpos" Y-Position: "ypos"
AnalogRead Volume Potentiometer Position
(Toggled) Current Button State Same as Last Button State?
YES
NO
Mute All Audio Output
Unmute All Audio Output
Set Volume Based on Volume-Potentiometer Position
Set Volume Based on Volume-Potentiometer Position
Remap volume-value to a range from 0 - 100
Constrain volume-value to a range from 0 - 100
Volume Currently Muted?
YES
NO
Print the current volume to the OLED display
Switch in "Top" Position?
YES
NO
Last Applied Effect = Current Effect?
YES
NO
Print current average Reverb delay time to OLED display
reverb(xpos)
lastEffect = 1 (Default = 0)
Print labels: "Reverb" and "Delay: " to OLED display
Switch in "Bottom" Position?
YES
NO
Last Applied Effect = Current Effect?
YES
NO
Print current # of "bits crushed", and sample rate for bitCrusher effect to OLED display
distortion(xpos, ypos)
lastEffect = 2 (Default = 0)
Print labels "Distortion: ", "Bits: ", and "Rate: " to OLED display
Last Applied Effect = Current Effect?
YES
NO
passThru()
lastEffect = 3 (Default = 0)
Print "No Effect" to OLED display

Figure 6-6: Arduino Software flowchart

The full code for this experiment it located in the appendix [Attachment 1].

The source code for this system consists of six functions listed below:

1.) void setup()
2.) void loop()
3.) void reverb()
4.) void distortion()
5.) void passThru()
6.) int avgDelayTime()

Functions 1 and two control the primary process flow in this system. Functions 3—6 are logically separate from the main process to increase system efficiency and organization.

6.4.1. Preprocessor Directives

The first set of instructions in this experiment is a series of preprocessor directives. The preprocessor directives in this experiment, from the top down, are used to:

1.) Include necessary libraries
2.) Define the classes of audio functions from the audio system design
3.) Define global variables
4.) Create object class for Debounce library
5.) Define an OLED I2C address
6.) Declare additional functions

6.4.2. Including Libraries

This experiment includes eight external libraries that must be installed prior to compilation. The table below specifies the libraries used in this experiment, and defines the library function.

Library	Function
`Bounce.h`	The Bounce.h library is a library created to de-bounce pushbutton switches [22]
`Audio.h`	The Audio.h library defines the objects (as classes) used as part of the TAL [23]
`Wire.h`	The Wire.h library is the library that allows communication with I2C devices [24]
`SPI.h`	The SPI.h library is the library that allows communication with SPI devices (MicroSD card) [25]
`SD.h`	The SD.h library allows for reading and writing to SD cards [26]
`SerialFlash.h`	The SerialFlash.h library provides low-latency, high-performance access to SPI Flash Memory (Teensy Audio Shield) [27]

`SSD1306Ascii.h`	The SSD1306Ascii.h library is a lightweight (unbuffered) character-only library for the SSD1306 OLED display [28]
`SSD1306AsciiWire.h`	The SSD1306Ascii.h library is a lightweight (unbuffered) character-only library for the SSD1306, used to communicate over the I2C bus [28]

Table 6-2: Summary Libraries Used in Experiment

6.4.3. OLED Ascii Library

The SSD1306 OLED display used in this experiment is a widely-distributed item. Several libraries exist to control this display.

This experiment utilizes a "lightweight" character-only library to control this display. The library is called "SSD1306Ascii" created by Bill Greiman [28].

This library does not buffer the data before sending it to the OLED. Display. While this means several features of more flexible library must be omitted, such as animations, and text-scrolling, an Ascii-only library requires very few system resources. Here, this saves memory for audio-related operations.

6.4.4. Defining Classes of Audio System Functions

The set of classes defined in the preprocessor directives for this experiment are automatically generated based off the audio system design created using the TAL GUI. Each class is assigned a unique identifier by which each audio object's functions may be controlled.

For instance, the gain of **mixer1** may be accessed by `mixer1.gain()`.

This list of automatically-generated classes and logical interconnects may be loaded *into* the TAL using the "Import" button to re-create the GUI of the audio system design.

6.4.5. Define Global Variables

The variables declared in the preprocessor directives are within the global scope of the program, and thus may be accessed in any function throughout the program.

Here, constants are used to define pin numbers of physical components in the circuit.

6.4.6. Create Object Class for Debounce

In order to de-bounce a pushbutton switch using the debounce library, it is necessary to first create a class that may be accessed within the operating functions of this program.

This experiment creates a class called "`StableBtn`."

6.4.7. Defined OLED I2C Address

Communicating with the I2C OLED, the OLED must be accessed by its hardware address. The SSD1306 OLED is a generic part, documented to have an address of either `0x3C` or `0x3D`. [29]

6.4.8. Declare Additional Functions

Lastly, before entering the `void setup()` function, it is necessary to declare additional custom functions used in the Sketch. Here, the only function returning a variable is the function `avgDelayTime()`. This function is used to compute the average delay time for the reverb effect.

6.4.9. Void Setup Function

The void setup function in this experiment is used to execute operations and defines presets that must only occur once.

For instance, this experiment uses the `void setup()` function to assign the pins for OLED display, selector switch, joystick, enable control of the Teensy Audio Shield: SGTL5000 chip, and lastly allocate system RAM for the audio shield.

6.4.9.1. Starting OLED Communication

The OLED library used in this experiment has two functions that are executed in the `void setup()` function:

```
oled.begin(OLED name, I2C address)
oled.setFont(font family)
```

Table 6-3: OLED library setup functions

The `oled.begin()` function initiates I2C communication with the OLED display via the I2C bus. The function `oled.setFont()` selects a default font to print the text.
When defining the pinMode function, here we define the pins used for digital I/O with the internal pull-up resistors on the Teensy 3.2 board. Use of the internal pullup resistors removes the need for external pullup resistors, keeping input pins HIGH (3.3V) be default. [30]

```
pinMode(pin, INPUT_PULLUP)
```

Table 6-4: Defining INPUT_PULLUP resistors

6.4.9.2. Allocating Audio Memory

In order to utilize the Teensy Audio board, it is necessary to allocate and reserve memory (RAM) used to store the audio data. Shown below is the audio memory function call syntax.

```
AudioMemory(numberBlocks)
```

Table 6-5: Allocating Teensy Audio Memory [32]

According to the Teensy Audio Board documentation: “The numberBlocks input specifies how much memory to reserve for audio data. Each block holds 128 audio samples, or approx

2.9 ms of sound. Usually an initial guess is made for numberBlocks and the actual usage is checked with `AudioMemoryUsageMax()`." [32]

This experiment sets aside 200 blocks of audio memory to maximize reverb delay duration, and save additional RAM for other system processes.

Teensy audio memory does not function dynamically. Once audio memory is reserved, no other system processes may use the RAM reserved for audio data—whether or not it is in active use. Likewise, the Teensy cannot dynamically request more memory by default. If the audio memory overflows, the program will encounter errors.

6.4.10. Void Loop Function

The void loop function iterates indefinitely for this experiment. See figure 6—5 for the full visual process flow for the void loop function.

Listed below is a chronological flow of operations within the void loop function:

1.) Read-in current selector switch position
2.) Read-in joystick position
3.) Read-in volume potentiometer position
4.) Check whether or not to mute the master audio-out, based on last loop iteration data
5.) Set master audio volume based on volume potentiometer data
6.) Remap and constrain master volume data to a range between 0 and 100
7.) Print "Volume" label, followed by volume value (converter to range of 0-10) to OLED display
8.) Determine desired selected audio effect

6.4.10.1. Read-in Selector Switch Position

Reading-in the selector switch position employs `digitalRead()` functions to read and store either a digital HIGH or LOW state. The selector switch position is stored as an integer value; either 0 for a digital LOW, or 1 for a digital HIGH.

6.4.10.2. Read-in Joystick Position

Reading-in the joystick position employs an `analogRead()` function, one for the x-axis, and one for the y-axis. Each value is stored independently as an integer value. The range of values for each axis is between 0 and 1023.

6.4.10.3. Read-in Volume Potentiometer Position

Reading-in the volume potentiometer position also employs an `analogRead()` function, storing the current potentiometer position as an integer between 0 and 1023.

6.4.10.4. Check “MUTE” Conditions

After reading in the HIDs, the system checks whether or not the conditions have been met to mute the master volume control.

Muting the master volume is controlled via momentary switch contained in the joystick package. Here, the Debounce library is incorporated to remove switch-bounce error when reading the switch press. Reading the momentary consists of three nested if-statements. The annotated code for this operation is copied from the source code, included below:

```
if (stableBtn.update())               //Button State Changed?
  {
    if (stableBtn.fallingEdge())      //Is there a falling-edge signal pulse?
    {
      if (currentPress != lastPress)  //Is the current toggle operation same as
                                      //the last toggle operation?
      {
        oled.setFont(Arial_bold_14);  //Set OLED font family
        oled.setCursor(65, 0);        //Set OLED cursor position
        oled.clearToEOL();            //Clear any present text to End Of Line (EOL)
        oled.print("| [MUTE]");       //Print “Mute” text
        oled.setFont(Arial14);        //Reset font family to default

        sgtl5000_1.muteLineout();     //Enable SGTL5000 Line-Out Mute function
        sgtl5000_1.muteHeadphone();   //Enable SGTL5000 Headphone Mute function
        currentPress++;               //Increment value stored in “currentPress”
        muted = true;                 //Set “muted” to true-state
      }

else if (currentPress == lastPress)      //Else, last toggle operation equal to
                                         //current toggle operation?
      {
        oled.setCursor(65, 0);           //Set OLED cursor position
        oled.clearToEOL();               //Clear any present text to End Of Line

        sgtl5000_1.unmuteLineout();      //Enable SGTL5000 LineOut Unmute function
        sgtl5000_1.unmuteHeadphone();    //Enable SGTL500 Headphone Unmute function
        lastPress++;                     //Increment value stored in “lastPress”
        muted = false;                   //Set “muted” to false-state
      }
    }
  }
```

Table 6-6: Volume Mute Operation

The entire mute operation is a toggle operation. Two integer variables independently store the last operation performed, and compare it against a variable in the current void loop iteration.

In the preprocessor directives, `currentPress` is initialized to 0, and `lastPress` is initialized to one. The void loop can only execute one of the two possible options for each iteration. When the button is pressed for the first time, the `lastPress` and `currentPress` already unequal values, and therefore the system mutes the master volume, incrementing `currentPress` as the final operation before exiting the conditional statement.

Therefore, next time the button is pressed, the conditional statement will find that `currentPress` and `lastPress` are now equal values, and the system unmutes the master volume, incrementing `lastPress` as the final operation before exiting the conditional statement.

Before printing to the OLED, first the cursor (seed location where to begin printing text) must be preset to a dedicated coordinate on the OLED matrix. The function setCursor is included below:

```
oled.setCursor(column, row) //Number of rows are integer multiples of eight pixels
```
Table 6-7: OLED setCursor function syntax

Before printing to the OLED display, it is necessary to clear any existing text at this location on the display. This OLED library includes a function shown below to clear all active pixels to the end of the line.

```
oled.clearToEOL()
```
Table 6-8: OLED clearToEOL function syntax

The figures below show what the OLED display looks like for each iteration of the Mute operation:

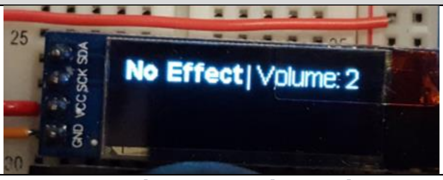
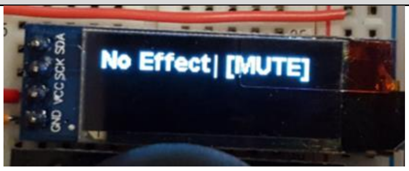
OLED Unmuted	OLED Muted
 No Effect| Volume: 2	 No Effect| [MUTE]

Figure 6-7: OLED Display, Mute and Unuted Text

6.4.10.5. Set Master Volume

Setting the master volume for this system is achieved through controlling the output of the SGTL5000 chip. Included below is an excerpt of code from the source:

```
//Master volume control

float vol = (float)volRaw / 1280.0;
sgtl5000_1.volume(vol);
```
Table 6-9: Master Volume Control Operation

This arithmetic operation limits the maximum volume to 80% to protect against hearing damage. The “raw” volume value (0-1023) is converted to a float (decimal) value for precision steps and a “soft” transition as the volume is swept through a predefined range.

The class `sgtl5000_1.volume()` is a function of the SGTL5000 chip on the Teensy audio board, capable of controlling the volume output of the LINE OUT and onboard headphone amplifier.

6.4.10.6. Remapping and Constraining Master Volume

Analog HID devices often read a range of values every cycle, and do not remain fixed at one value or another. This "over-precision" of analog control devices often creates "jumping" values that can deviate as much as 20 steps in a range of 0-1023 values as observed in this experiment.

To print the volume level to the OLED display, first the value for the current volume is must be converted from a range of 0—1023, to 0—100. This is done using the `map()` and `constrain()` functions.

Here, the `map()` function is used in an untraditional way: 1023 steps are fed to `map()` as the range from which to translate. However, the `map()` function only executes and returns integer values. Notice, this experiment defines vol as a float value. Therefore, the map function treats the range from 0—0.8 (min to max for vol) as a 0—1023 steps. This gives each "step" of the volume a much wider range of raw values before mapping to the next value in 0-100 steps. [32]

In other words, here the `map()` function behaves like a range buffer, giving "cushion" to the raw volume values, and solidifying each step in the range of 0—100, effectively eliminating "jumping" values.

6.4.10.7. Printing Volume Data to OLED Display

Once converted a conditional statement is called to determine if the system is currently muted, or unmuted. If the system is not muted, volume data is printed to the OLED using the code excerpt included below.

```
if (muted == false)
  {
    //OLED printing info to second line of screen
    oled.setCursor(65, 0);       //Init cursor to volume location
    oled.print("| Volume: ");     //Volume label

    //This arithmetic operation converts the 0-80% volume range to an integer volume
    //between 0 and 10:
    oled.print((int)(vol * 137.5));
    oled.print(" ");     //Adding a blank space following the changing values to
                         //prevent "stuck pixels"
  }
```

Table 6-10: Operation, printing volume data to OLED display

Printing the volume data to the OLED display utilizes the `setCursor()`, and `print()` functions.

The arithmetic operation performed when printing the volume value converts the float volume-value to an integer after limiting the volume range from 0-10 for logical perception when changing volume.

The range of 0—10 is counted as not *ten* but *eleven* steps. After being mapped to a range from 0—100, the new, current *maximum* value for vol can be 0.08.

$$0.08 * \boldsymbol{coeff} = 11\ steps$$
$$\boldsymbol{coeff} = 137.5$$

Thus, returning a volume range from 0—10 is possible after finally converting the float value to an integer.

An additional `oled.print()` function is included to "clear out" any remaining pixels from a previous value.

6.4.10.8. Determine Desired Audio Effect

The last step in the void loop function is to determine which audio effect is currently selected by the user. This step is a series of conditional if-statements.

The three-position selector switch has a variable to store the current status of each pin; whether it is currently reading a digital HIGH or digital LOW.

The first conditional statement is whether the variable selectTop is LOW or ("Active" given each input pin in tied HIGH internally, by default). This switch position activates the "reverb" function.

The next conditional statement is whether the variable selectBtm is LOW. This switch position activates the "distortion" audio effect.

The last conditional statement is the default should neither selectTop or selectBtm return true. This last default condition is the physical middle position on the selector switch. This position activates the audio "passThru" function to pass all audio signals directly through the system without any modulation.

Within each parent conditional-if statement is a nested conditional if-statement to determine whether or not this current, selected effect is equal to the effect that was active during the last voice loop iteration.

This degree of information granularity allows the system to determine whether or not to re-draw the labels associated with each effect.

This additional, nested if-statement was added after early experimentation due to extreme flickering as a result of re-drawing all labels on the OLED display with each loop iteration. Using this additional if-statement instructs the system to only redraw the effect-parameters with each void loop iteration, completely eliminating display-refresh flicker.

The last two operations in each effect conditional statement is a call to an external function for that effect, and storing an integer value in the variable lastEffect. This variable represents the effect selected during the last iteration of the void loop.

The reverb, distortion, and passThru are each external functions to the void loop function. The reverb, and distortion effects must be passed the current reading of the joystick position. The table below shows the function calls for the reverb and distortion effects. The variables “xpos” and “ypos” hold the current values for the joystick position, on a scale from 0—1023.

`reverb(xpos);`
`distortion(xpos, ypos);`

Table 6-11: reverb() and distortion() Function calls

6.4.11. Void Reverb Function

The void reverb function only has one control parameter: delay time. Therefore, the only value read-into reverb is the joystick x-position.

The reverb function is “activated” by setting the gains of channels 2 and 3 of the final-stage mixer (mixer1) to zero (full attenuation).

Channels 0 and 1 control the mix between the amount of reverb to sum with the original signal. Similar to a live-music show, the original signal should receive gain bias to produce a realistic-sounding reverb effect.

Mixers 2, 3, and 4 control the gain at each stage of signal-summation. The values selected for each mixer is based on personal preference and may be altered to “tune” the system.
The syntax for the mixer gain function is shown in the table below:

`mixer.gain(channel, gain)`

Table 6-12: Mixer gain function syntax

Within the reverb function is a Biquad filter, setup as low-pass filter. The Biquad filter in the TAL has a predefined function to configure the filter for low-pass. The syntax for this function is included below:

`setLowpass(stage, frequency, Q);`

Table 6-13: Biquad, low-pass filter syntax

For the function, stage controls the number of stages to cascade through the filter (0—3), frequency controls the corner frequency of the filter and “Q” controls the Q-shape of the wave (recommended to be below 0.707). Like the mixers, the low-pass filter may be configured according to preference. Here, the values selected are:

- Stage = 3
- Frequency = 4000 (Hz)
- Q = 0.699

The reverb effect contains two conditional if-statements.

The first if-statement checks the value of reverb set by the user based on the input. The parameters for the reverb effect require additional tuning to remove system noise. Therefore, this statement was added to completely attenuate the reverb effect if the value of reverb is set to be less than 50 (based on a range from 0—1023).

The last operation performed in the reverb effect is a for-loop used to dynamically set the delays for each channel of the delay-tap object.

The irrational multipliers for each delay-tap line is based on a table of values from a $\log_{10}$ curve. These values ensure that no delay-line is an integer-multiple of another delay-tap channel. The delay multipliers are stored in a 4x1 global array called `pauses[]`.

`pauses[]` Array Values:			
0.602	0.699	0.788	0.850

Table 6-14: Reverb pauses[] array; delay time multipliers

Since the first four channels of the delay object are used for feedback, and the other four channels of the delay are immediately summed with the real-time signal output, only four multiplier values are necessary.

The entire for loop for this dynamic-delay-setting is included in the table below:

```
//Applying pauses[] coefficients to delay lines
  for (int i = 0; i < 8; i++)
  {
//Setting delay durations for first "four" (of eight) channels (0-3)
    if (i < 4)
    {
      delay1.delay(i, val1 * pauses[i]);
    }

//Setting delay durations for second "four" (of eight) channel (4-7)
```

```
    else if (i >= 4)
    {
      delay1.delay(i, val1 * pauses[i - 4]);
    }
  }
```
Table 6-15: Setting dynamic reverb delay lines

6.4.12. Void Distortion Function

Similar to the reverb effect, the distortion (also called “bitcrusher” in the TAL) function is controlled by selecting a value for the final-stage gain (mixer1) on channels 2 and 3, and fully attenuating (zeroing) the values of gain for channels 0 and 1.

The distortion effect reads-in the values stored for the joystick x and y position. Here, the x-position is used to control the number of bits crushed, and the y position is used to control the sample rate of the effect.

Before sending the values to the bitcrusher effect, they must first be re-mapped, and constrained to the range of accepted values for each function.

The number of bits crushed for the bit crusher effect must be between 0—16. When zero bits are crushed, the effect behaves like a pass-through.

The sample rate for the bit crusher effect must be between 0 and 44100. When the sample rate is close to 44100, the effect behaves like a pass-through.

The `map()` and `constrain()` functions are used to re-map the x and y joystick values to a range of 0—16, and 0—44100, respectively.

Given the relatively short-range of x-values for the number of bits crushed, the input values were re-assigned to an output following an exponential curve. For a linear, input, following an exponential curve gives the user greater control over the number of bits crushed at the low end as the signal becomes increasingly “unusable” as the number of bits crushed approaches the maximum value of 16.

The figure on the on the following page shows the curve graphed on an X-Y axis, where the x-axis represented the joystick x-input, and the y-axis represents the re-assigned bits-to-be-crushed function input.

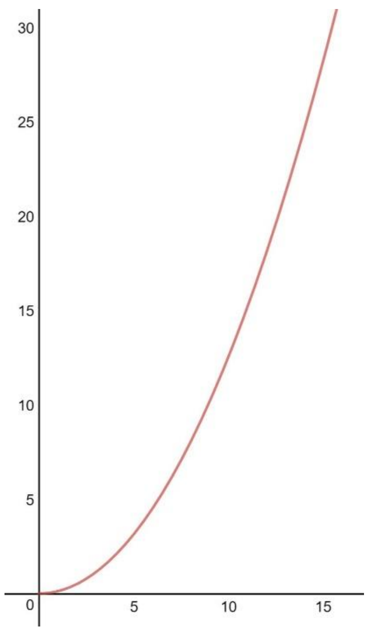


Figure 6-8: Exponential Distortion-Effect Control Curve

The table below includes the syntax for the bitcrusher effect:

`bits(xcrushBits);`
`sampleRate(xsampleRate);`

Table 6-16: Bitcrusher Syntax

6.4.13. Pass-Thru Function

The pass-thru function in this system only has one function: control the signal attenuation at the final stage mixer (mixer1). To pass-through the original signal, mixer channels 0—2 are fully attenuated (set to zero gain). Channel 3 of the final stage mixer receives the original source signal from mixer 5 (refer to audio system flow chart Figure 5—1) and send it to the I^{2}S output. This function is called when the system selector switch is neither in the top or bottom position, and functions as a default-state.

6.4.14. Int avgDelayTime Function

The last function in this system is the only function used to return a value. The purpose of the avgDelayTime function is to calculate the "average" delay time based on the constant delay-line multipliers, after multiplied by the current joystick position to derive a value measured in milliseconds. This computed value is intended for use to show the average reverb delay time on the OLED display.

The avgDelayTime function executes the same arithmetic performed in the reverb function. However, instead of calculating individual delay values, this function computes the average of all values stored in the `pauses[]` array, and multiplies that average value by the current joystick position value. The result is a delay-average for the four-values stored in pauses[], measured in milliseconds.

Many steps could be saved by computing the average of all the values in pauses[] by hand. However, in doing so, one would lose the ability to “tune” this system according to preference without manually calculating the average delay time with each alternation.

The table below includes the entire avgDelayTime annotated function from the source code:

```
int avgDelayTime(int joyPosition)
{
  float sum = 0;      //Summing var
  int delayTime = 0;  //Time of reverb delay measured in ms

  for (int i = 0; i < 3; i++)   //Summing of all delay multipliers in pauses array
  {
    sum += pauses[i];    //”Sum” adds together adjacent values stored in pauses[]
  }

  sum = sum / 4;        //Taking average by division by four

//Multiplying avg. delaytime coeff. by joystick input, deriving delay time in ms
  delayTime = sum * joyPosition;
  return (int)delayTime;    //Returning delay time
}
```

Table 6-17: avgDelayTime Function

The first operations define the local variables to be used. Next, a for-loop is constructed to sum each value in the delay-multiplier array pauses[]. Then the total sum of the values stored in pauses is divided by the size of the array, 4 to compute the average. Lastly, the delay time is calculated by multiplying the average delay-coefficient by the current value of the joystick. Then the value is converter from a float to an integer, and returned back to the function call.

7. Analysis and Testing

6.5. Testing OLED Display

Before audio effect testing, first the OLED display was tested to ensure the display functioned as expected with the OLED library.

The SSD1306Ascii library includes an example Sketch called “FontSamplesWire” located under: File → Examples → SSD1306Ascii → FontSamplesWire in the Arduino IDE. The OLED Test Sketch is also included in the appendix as [Attachment 3].

The only modification required to this example sketch is re-defining the OLED dimensions on line 68. Here, the dimensions for the OLED are preset as: 128x64. This experiment uses a 128x32 OLED display. Therefore, this line must be changed as shown in the table below

Example FontSamplesWire, Line 68:	Modified FontSamplesWire, Line 68
`oled.begin(&Adafruit128x64, I2C_ADDRESS);`	`oled.begin(&Adafruit128x32, I2C_ADDRESS);`

Table 7-1: Modified OLED Example Sketch

After programming, the 128x32 OLED display should cycle through a variety of fonts shown by the font title, and alphabet.

The figure below shows the system running the OLED-test Sketch, displaying text using the font "ZevvPeep."

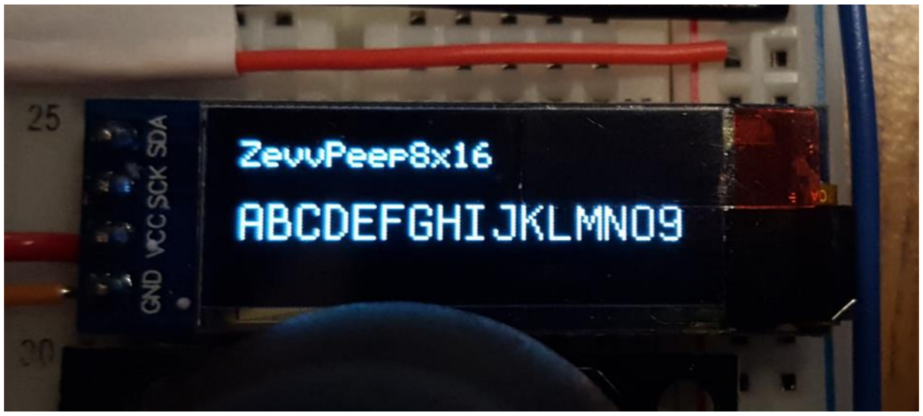


Figure 7-1: OLED Functionality Testing

6.6. Testing Reverb Effect

Testing the reverb effect required the use of an oscilloscope a Keysight function generator. The system was tested using the following settings for input:

<u>Function Generator:</u>
Waveform = Siusoid
Frequency = 1 kHz
Amplitude = $2V_{pp}$

Before testing the reverb effect, first, the system was tested using a Pass-through effect. The following figure shows a screen-capture of the oscilloscope measuring the signal with the system operating in pass-through.

For this experiment, measurements of the input signal are captured on channel one. System output signals are measured on channel two.

An additional measurement is included on the scope to measure the delay through introduced by the entire audio system form input to output. Here the, signal latency is approximately 145 microseconds.

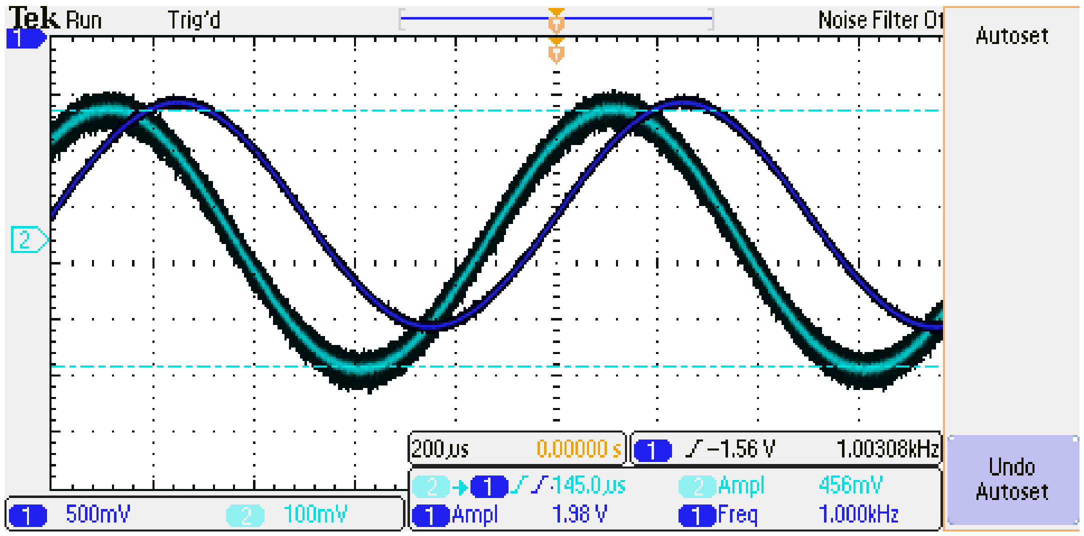


Figure 7-2: Waveform, Testing Reverb Effect

The next figure shows a capture of the system when the reverb effect is fully-engaged for a maximum reverberation delay.

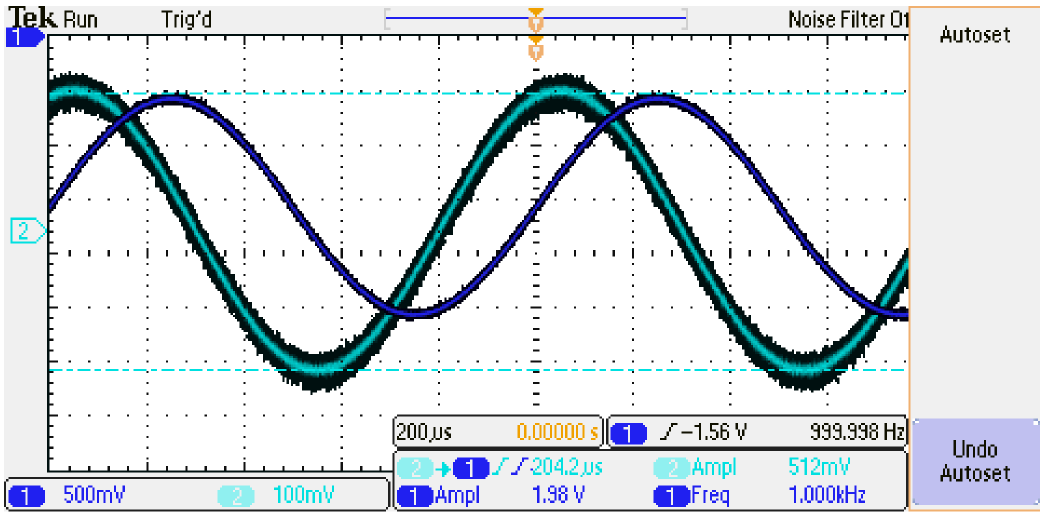


Figure 7-3: Waveform, Testing Reverb Effect, Maximum Delay

With the reverb effect fully-engaged, the delay changes to a measured 200 microseconds. Viewing the system at a maximum latency reveals apparent maximum reverb-delay to be approximately 530ms (as shown in the OLED-display figure):

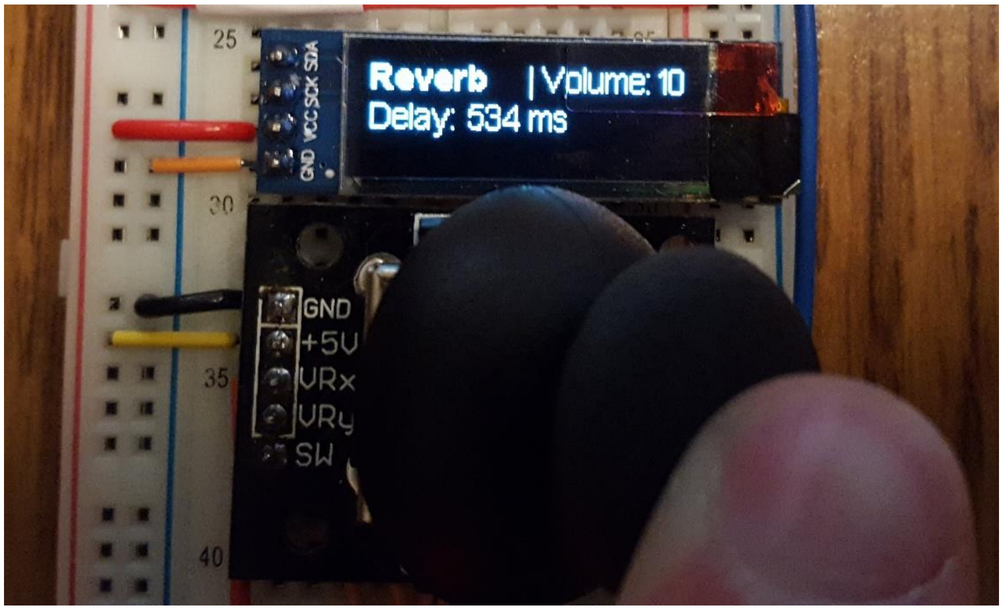


Figure 7-4: Testing Maximum Reverb Delay

While there appears to be a discrepancy in delay. The true delay must be calculated differently. The delay shown on the oscilloscope only returns the delay of the first (shortest-duration delay) when compared to the input signal. Additionally, the delayed wave is only represented by approximately 40% of the final output signal given the mixer-gain configuration for this experiment.

The shortest duration delay stored in the array pauses[] is the value 0.602. Additionally, the oscilloscope measurement of "delay" is representative of the amount (or "duration") of delay for each cycle. Here, the cycle is 1 kHz, or 1000 cycles per second.

This data is an approximation of the duration of delay for the shortest signal pulse through the system. This equation does not account for the variables introduced by feedback, and secondary reflections.

Full signal analysis exceeds the scope of this experiment. However, further signal integrity analysis may be conducted using this setup in conjunction with statistical methods of delay estimation, such as complex cross-correlation [33].

6.7. Testing Distortion Effect

To test the distortion effect, an oscilloscope is connected to the Teensy on-board DAC. This testing will be conducted by generating a sinusoidal wave internally, using the Teensy 3.2 and TAL.

Shown here is the audio system constructed to generate a sinusoid for output on the Teensy DAC:

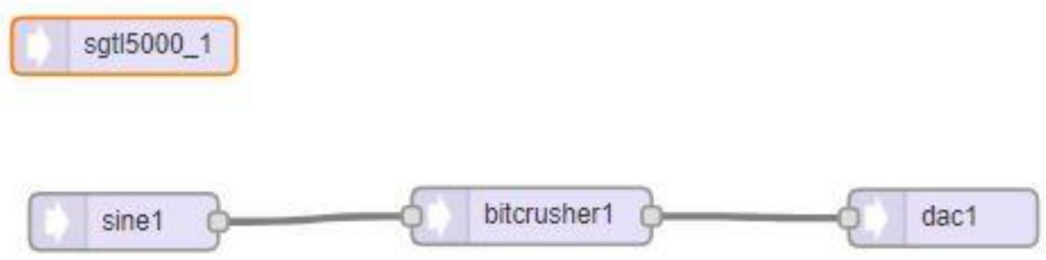


Figure 7-5: Distortion Effect Test

The Sketch Code for the distortion effect test is listed in the appendix [Attachment 2].

Here, a sine wave is defined with the following parameters:

<u>Internally-Generated</u> <u>Sine Wave</u>

Amplitude = 1 (maximum Teensy output ~ 1.3V)

Frequency = 1kHz

Connecting a DSO138 oscilloscope to the Teensy DAC on pin A14 shows the internally generated sine wave, shown below:

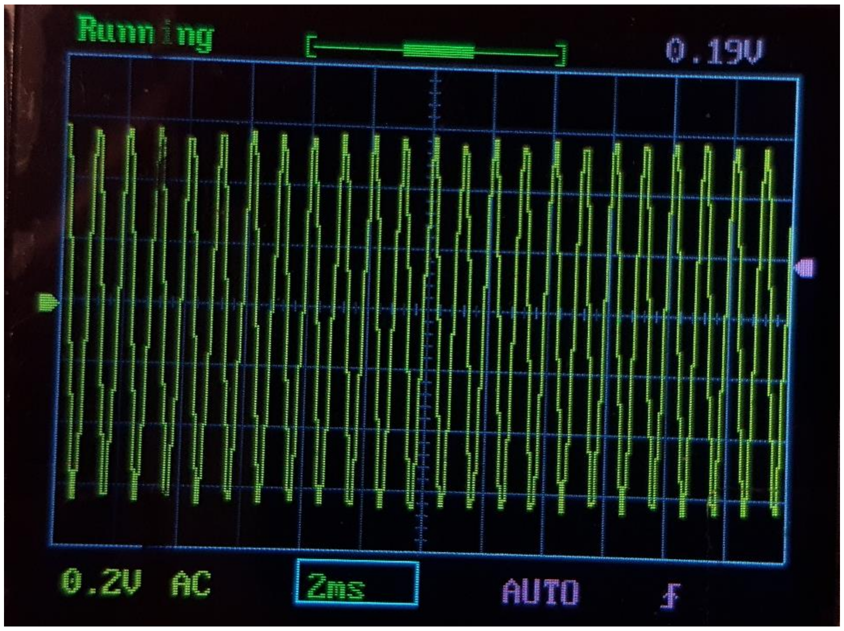


Figure 7-6: Teensy-generated 1kHz Sine Wave

Setting the distortion (bitcrusher) effect with the follow parameters produced a waveform shown in the next figure.

Bitcrusher Parameters:

Number of bits crushed = 0

Sample Rate = 20,000

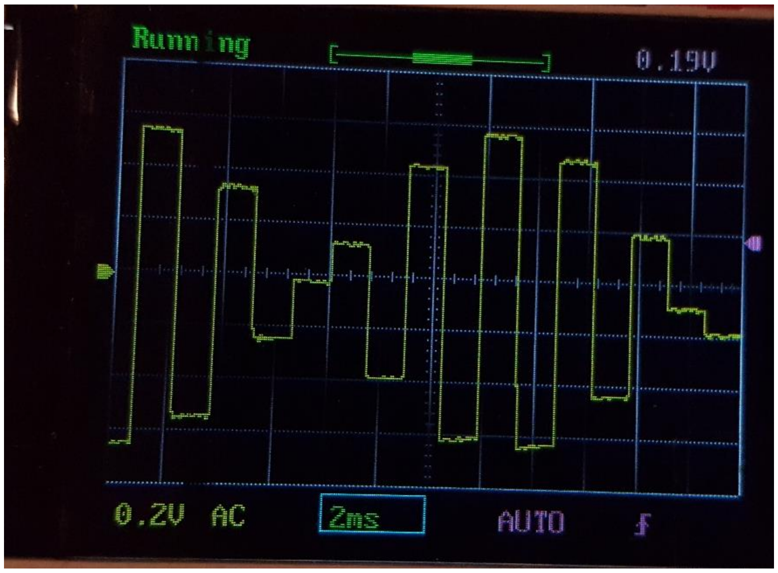


Figure 7-7: Distortion Effect, Sample Rate = 20k

Changing the distortion effect to the follows parameters produced a clipped-wave shown in figure below:

Bitcrusher Parameters:

Sample Rate = 44100

Number of Bits Crushed = 3

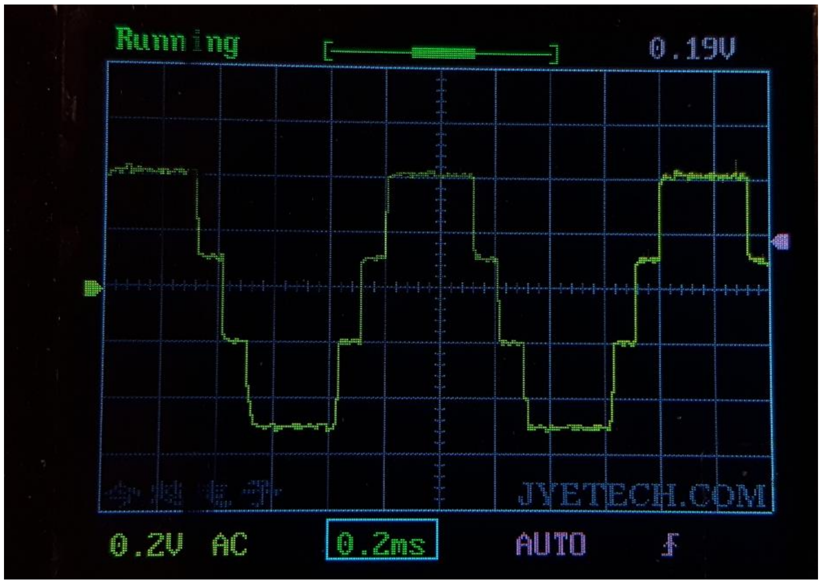


Figure 7-10: Distortion Effect, 2 bits crushed

The additional "steps" shown in the waveform are produced by the resolution the Teensy 3.2 DAC, as well as a resolution of the DSO Oscilloscope used for measurements.
When the distortion effect is fully-engaged at extreme parameters:

<u>Bitcrusher Parameters:</u>
Sample Rate = 1
Number of bits crushed = 16

The disotrtion effect produces a square-wave shown in the figure below:

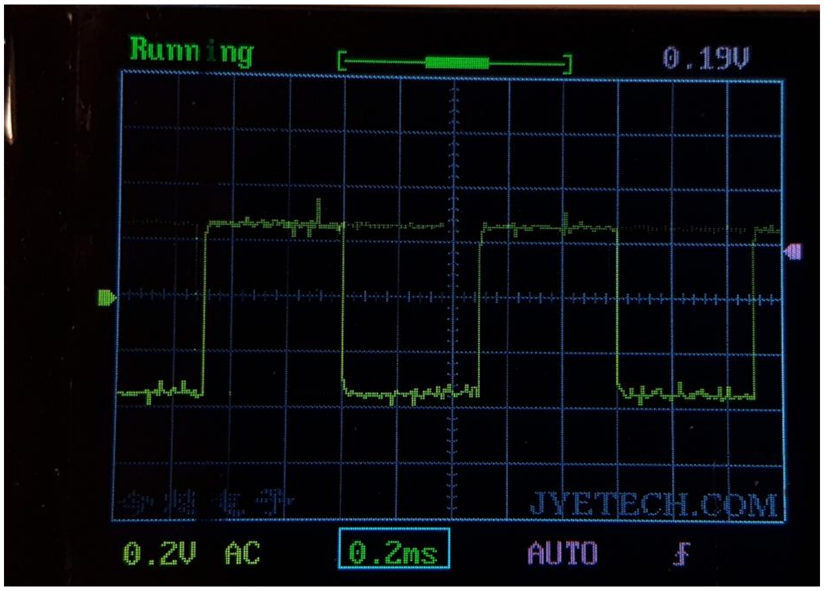


Figure 7-13: Distortion Effect Extreme Values

The distortion effect is not currently in a "usable" state as documented in the TAL. The distortion-sound is extreme, and contains a great deal of signal noise.

This effect was selected for this experiment given its extreme properties. While currently unusable for most sound-related applications, the characteristics of this effect are very stark in comparison to other effects, even in comparison to the reverb.

8. Final Evaluation

This experiment as an entry-level DSP system met the goals and objectives established in the scope, as defined for this experiment. This system is capable of reading-in analog audio signals in, converting the analog audio signal into digital data, applying "effect" transformations in "real-time" with unperceivable latency, and outputting the modulated audio signal to an analog device.

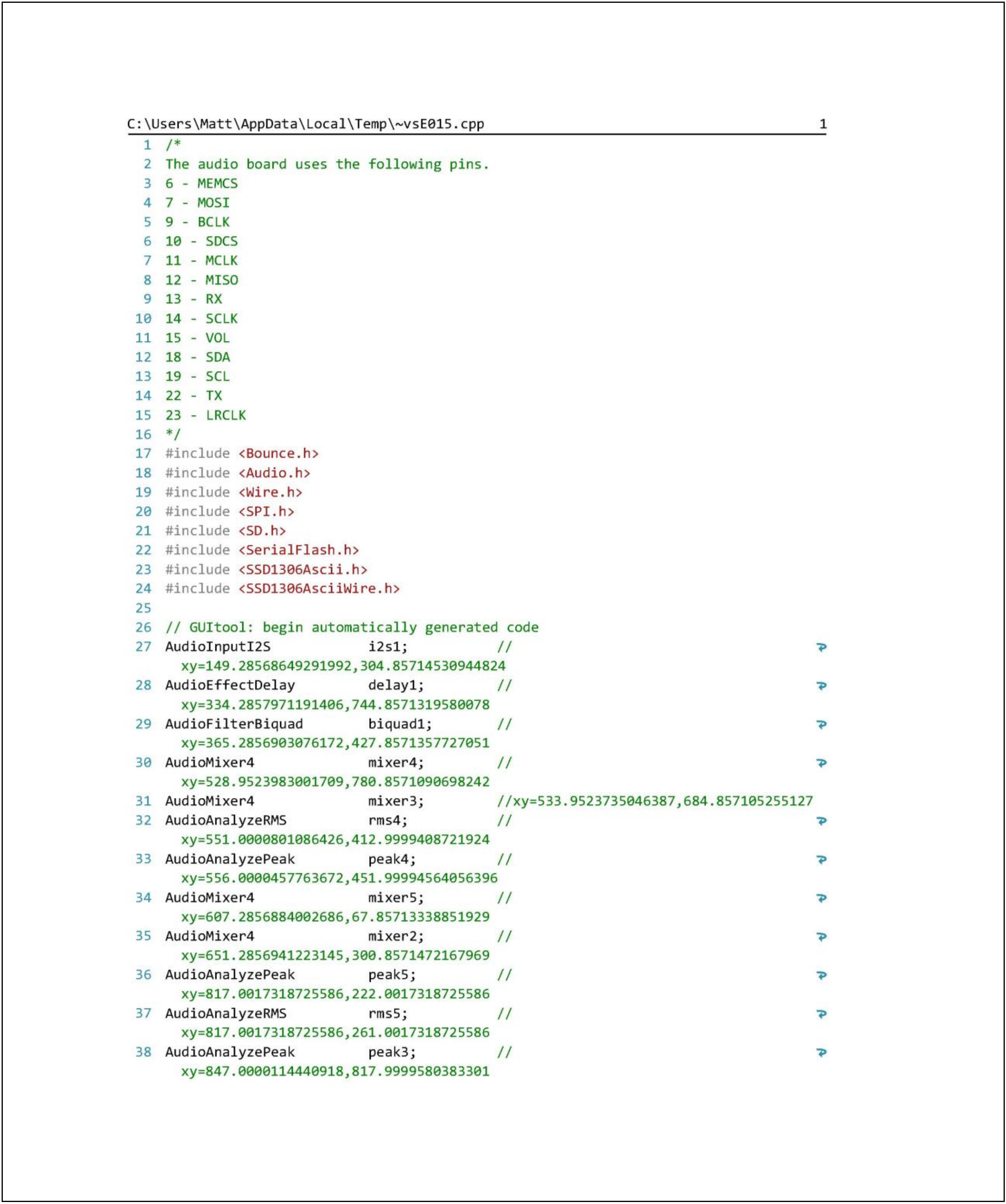
The code used in this experiment separated processes into logical sectors. Operations that only needed executed at a desired time, or following a specific action were separated into individual functions. Moreover, variables were carefully defined within proper scope, at the beginning of the Sketch. This organizational structure gives the Sketch flexibility for system tuning and troubleshooting without modification of core-operation functionality.

The audio system described in this experiment is adequate to process line-level audio with minimal system latency. However, such a system is inefficient for use on a larger-scale. Lack of available RAM significantly limits the system capability in how much audio-information can be dynamically stored. Lack of sufficient RAM is the primary reason why the "Reverb" object in the TAL cannot be used on the Teensy 3.2 hardware.

However, for applications that do not require delaying signals, the Teensy 3.2 and Audio Shield, given the 16-bit 44.1 kHz audio quality, are extremely versatile tools for developing high-resolution, prototype DSP-systems at low cost.

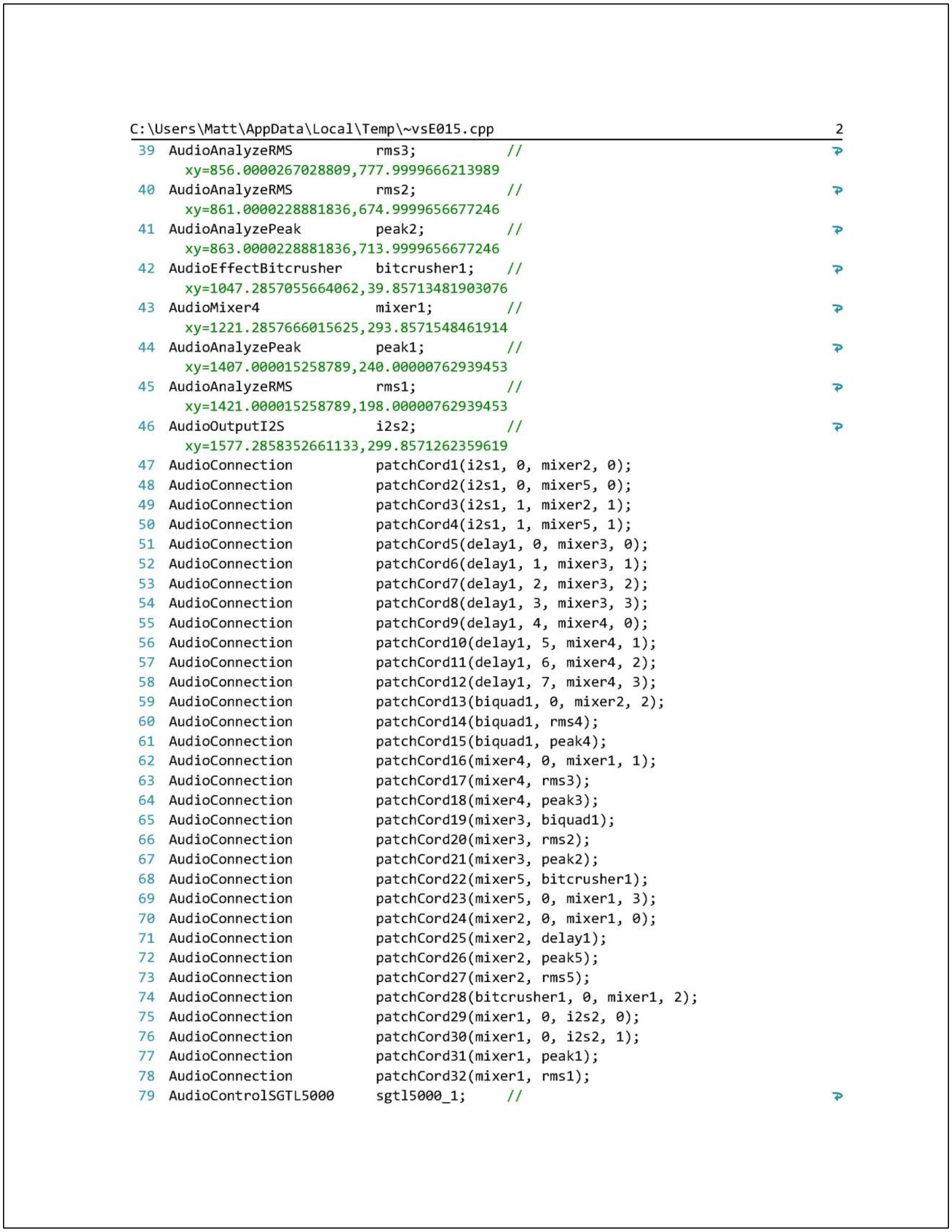
9. Attachments

[Attachment 1] Audio System Master Sketch Code



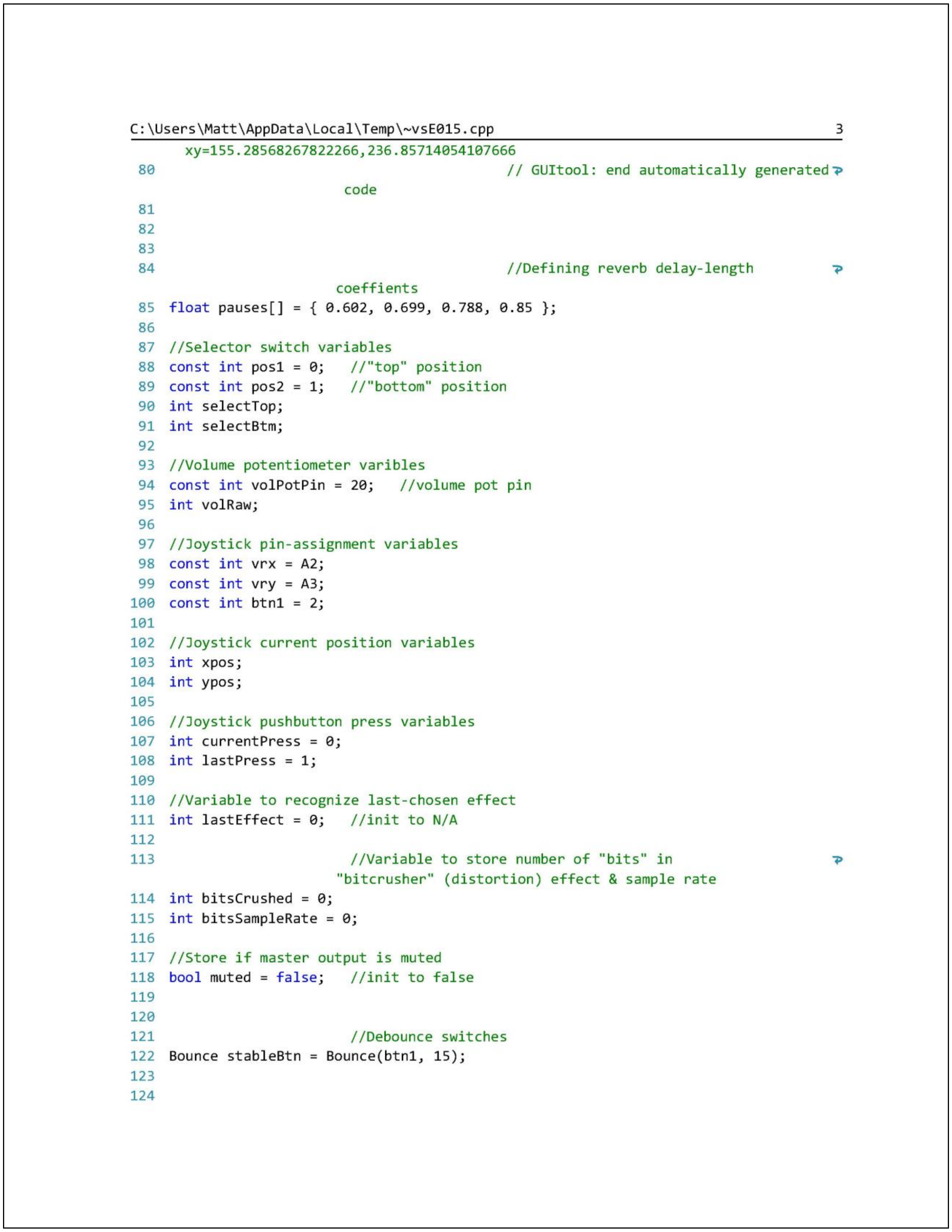
C:\Users\Matt\AppData\Local\Temp\~vsE015.cpp 1

```
/*
The audio board uses the following pins.
6 - MEMCS
7 - MOSI
9 - BCLK
10 - SDCS
11 - MCLK
12 - MISO
13 - RX
14 - SCLK
15 - VOL
18 - SDA
19 - SCL
22 - TX
23 - LRCLK
*/
#include <Bounce.h>
#include <Audio.h>
#include <Wire.h>
#include <SPI.h>
#include <SD.h>
#include <SerialFlash.h>
#include <SSD1306Ascii.h>
#include <SSD1306AsciiWire.h>

// GUItool: begin automatically generated code
AudioInputI2S            i2s1;           //xy=149.28568649291992,304.85714530944824
AudioEffectDelay         delay1;         //xy=334.2857971191406,744.8571319580078
AudioFilterBiquad        biquad1;        //xy=365.2856903076172,427.8571357727051
AudioMixer4              mixer4;         //xy=528.9523983001709,780.8571090698242
AudioMixer4              mixer3;         //xy=533.9523735046387,684.857105255127
AudioAnalyzeRMS          rms4;           //xy=551.0000801086426,412.9999408721924
AudioAnalyzePeak         peak4;          //xy=556.0000457763672,451.99994564056396
AudioMixer4              mixer5;         //xy=607.2856884002686,67.85713338851929
AudioMixer4              mixer2;         //xy=651.2856941223145,300.8571472167969
AudioAnalyzePeak         peak5;          //xy=817.0017318725586,222.0017318725586
AudioAnalyzeRMS          rms5;           //xy=817.0017318725586,261.0017318725586
AudioAnalyzePeak         peak3;          //xy=847.0000114440918,817.9999580383301
```



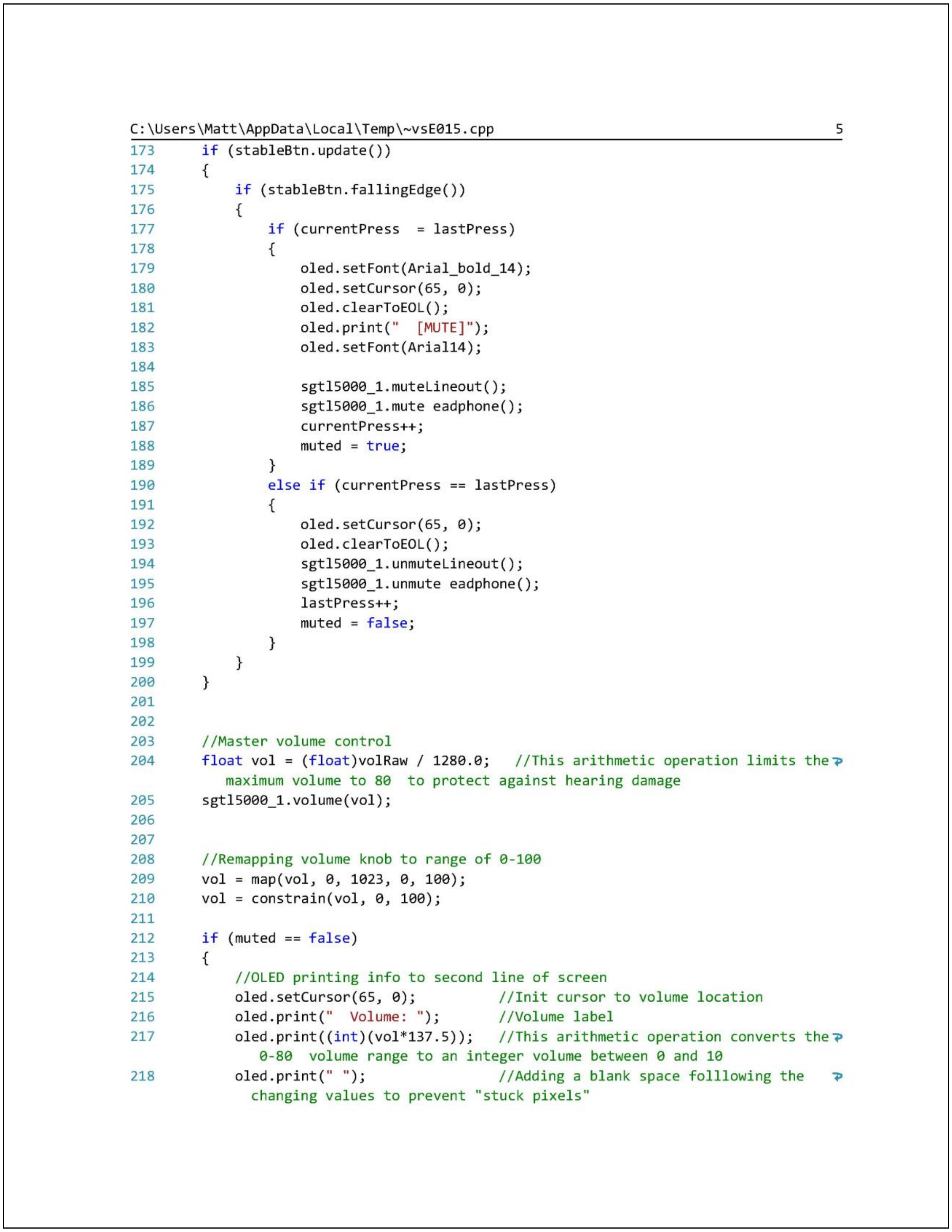
C:\Users\Matt\AppData\Local\Temp\~vsE015.cpp 2

```
AudioAnalyzeRMS          rms3;           //xy=856.0000267028809,777.9999666213989
AudioAnalyzeRMS          rms2;           //xy=861.0000228881836,674.9999656677246
AudioAnalyzePeak         peak2;          //xy=863.0000228881836,713.9999656677246
AudioEffectBitcrusher    bitcrusher1;    //xy=1047.2857055664062,39.85713481903076
AudioMixer4              mixer1;         //xy=1221.2857666015625,293.8571548461914
AudioAnalyzePeak         peak1;          //xy=1407.000015258789,240.00000762939453
AudioAnalyzeRMS          rms1;           //xy=1421.000015258789,198.00000762939453
AudioOutputI2S           i2s2;           //xy=1577.2858352661133,299.8571262359619
AudioConnection          patchCord1(i2s1, 0, mixer2, 0);
AudioConnection          patchCord2(i2s1, 0, mixer5, 0);
AudioConnection          patchCord3(i2s1, 1, mixer2, 1);
AudioConnection          patchCord4(i2s1, 1, mixer5, 1);
AudioConnection          patchCord5(delay1, 0, mixer3, 0);
AudioConnection          patchCord6(delay1, 1, mixer3, 1);
AudioConnection          patchCord7(delay1, 2, mixer3, 2);
AudioConnection          patchCord8(delay1, 3, mixer3, 3);
AudioConnection          patchCord9(delay1, 4, mixer4, 0);
AudioConnection          patchCord10(delay1, 5, mixer4, 1);
AudioConnection          patchCord11(delay1, 6, mixer4, 2);
AudioConnection          patchCord12(delay1, 7, mixer4, 3);
AudioConnection          patchCord13(biquad1, 0, mixer2, 2);
AudioConnection          patchCord14(biquad1, rms4);
AudioConnection          patchCord15(biquad1, peak4);
AudioConnection          patchCord16(mixer4, 0, mixer1, 1);
AudioConnection          patchCord17(mixer4, rms3);
AudioConnection          patchCord18(mixer4, peak3);
AudioConnection          patchCord19(mixer3, biquad1);
AudioConnection          patchCord20(mixer3, rms2);
AudioConnection          patchCord21(mixer3, peak2);
AudioConnection          patchCord22(mixer5, bitcrusher1);
AudioConnection          patchCord23(mixer5, 0, mixer1, 3);
AudioConnection          patchCord24(mixer2, 0, mixer1, 0);
AudioConnection          patchCord25(mixer2, delay1);
AudioConnection          patchCord26(mixer2, peak5);
AudioConnection          patchCord27(mixer2, rms5);
AudioConnection          patchCord28(bitcrusher1, 0, mixer1, 2);
AudioConnection          patchCord29(mixer1, 0, i2s2, 0);
AudioConnection          patchCord30(mixer1, 0, i2s2, 1);
AudioConnection          patchCord31(mixer1, peak1);
AudioConnection          patchCord32(mixer1, rms1);
AudioControlSGTL5000     sgtl5000_1;     //
```



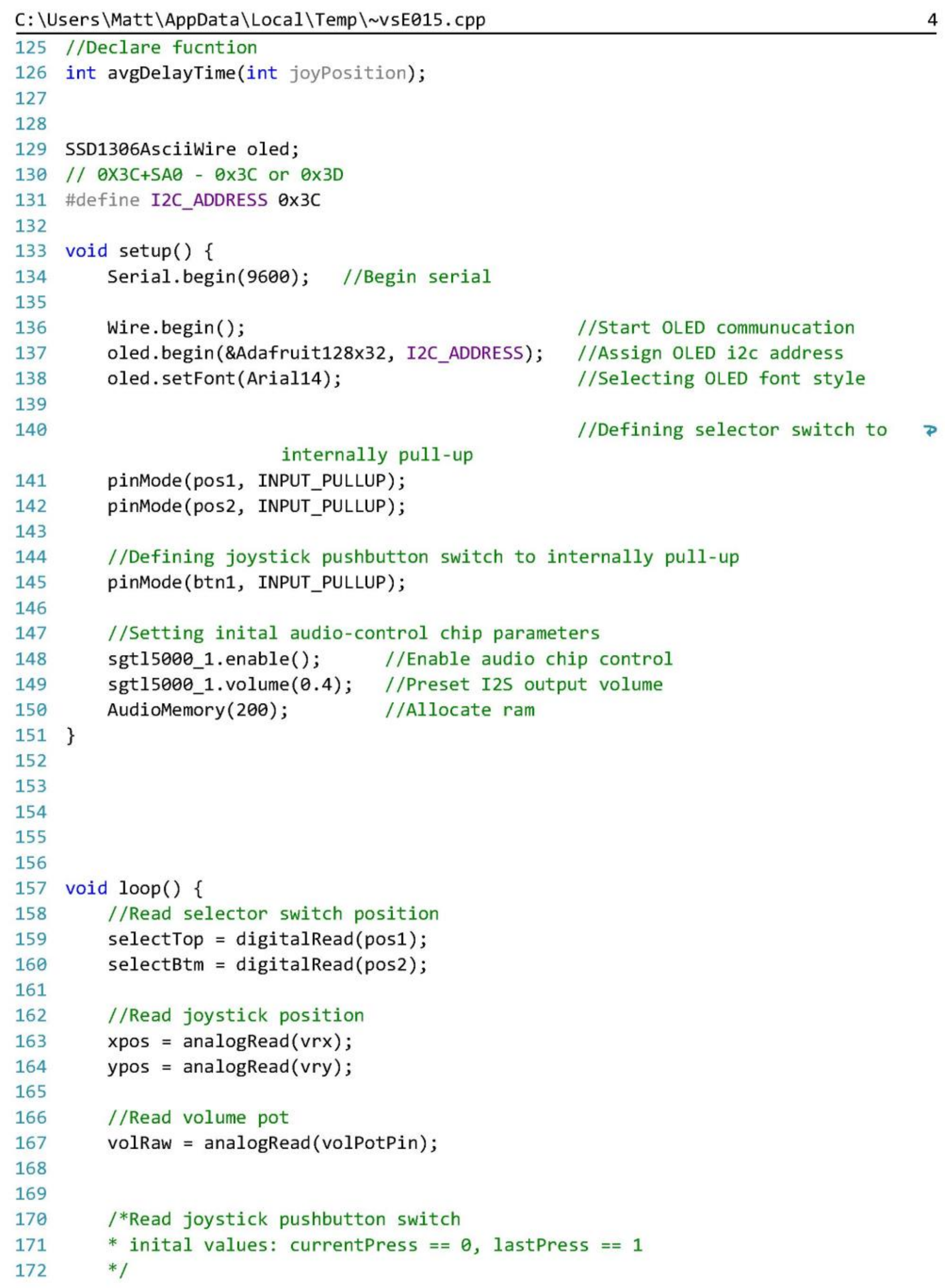
C:\Users\Matt\AppData\Local\Temp\~vsE015.cpp 3

```cpp
xy=155.28568267822266,236.85714054107666
                                        // GUItool: end automatically generated code



                                        //Defining reverb delay-length coeffients
float pauses[] = { 0.602, 0.699, 0.788, 0.85 };

//Selector switch variables
const int pos1 = 0;   //"top" position
const int pos2 = 1;   //"bottom" position
int selectTop;
int selectBtm;

//Volume potentiometer varibles
const int volPotPin = 20;   //volume pot pin
int volRaw;

//Joystick pin-assignment variables
const int vrx = A2;
const int vry = A3;
const int btn1 = 2;

//Joystick current position variables
int xpos;
int ypos;

//Joystick pushbutton press variables
int currentPress = 0;
int lastPress = 1;

//Variable to recognize last-chosen effect
int lastEffect = 0;   //init to N/A

                      //Variable to store number of "bits" in "bitcrusher" (distortion) effect & sample rate
int bitsCrushed = 0;
int bitsSampleRate = 0;

//Store if master output is muted
bool muted = false;   //init to false


                      //Debounce switches
Bounce stableBtn = Bounce(btn1, 15);


```



C:\Users\Matt\AppData\Local\Temp\~vsE015.cpp 5

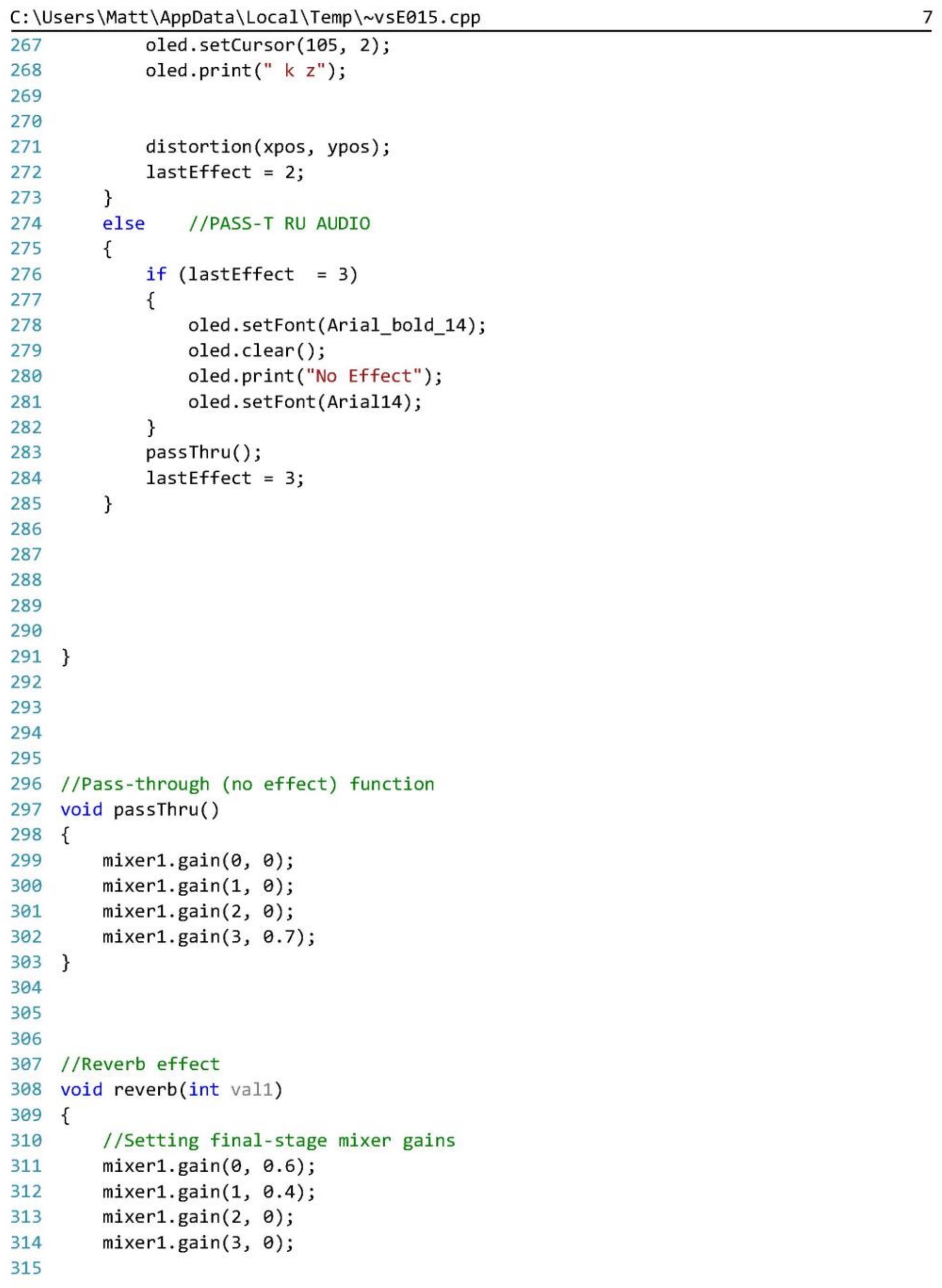
```cpp
    if (stableBtn.update())
    {
        if (stableBtn.fallingEdge())
        {
            if (currentPress  = lastPress)
            {
                oled.setFont(Arial_bold_14);
                oled.setCursor(65, 0);
                oled.clearToEOL();
                oled.print("  [MUTE]");
                oled.setFont(Arial14);

                sgtl5000_1.muteLineout();
                sgtl5000_1.mute eadphone();
                currentPress++;
                muted = true;
            }
            else if (currentPress == lastPress)
            {
                oled.setCursor(65, 0);
                oled.clearToEOL();
                sgtl5000_1.unmuteLineout();
                sgtl5000_1.unmute eadphone();
                lastPress++;
                muted = false;
            }
        }
    }


    //Master volume control
    float vol = (float)volRaw / 1280.0;   //This arithmetic operation limits the maximum volume to 80  to protect against hearing damage
    sgtl5000_1.volume(vol);


    //Remapping volume knob to range of 0-100
    vol = map(vol, 0, 1023, 0, 100);
    vol = constrain(vol, 0, 100);

    if (muted == false)
    {
        //OLED printing info to second line of screen
        oled.setCursor(65, 0);          //Init cursor to volume location
        oled.print("  Volume: ");       //Volume label
        oled.print((int)(vol*137.5));   //This arithmetic operation converts the 0-80  volume range to an integer volume between 0 and 10
        oled.print(" ");                //Adding a blank space folllowing the changing values to prevent "stuck pixels"
```



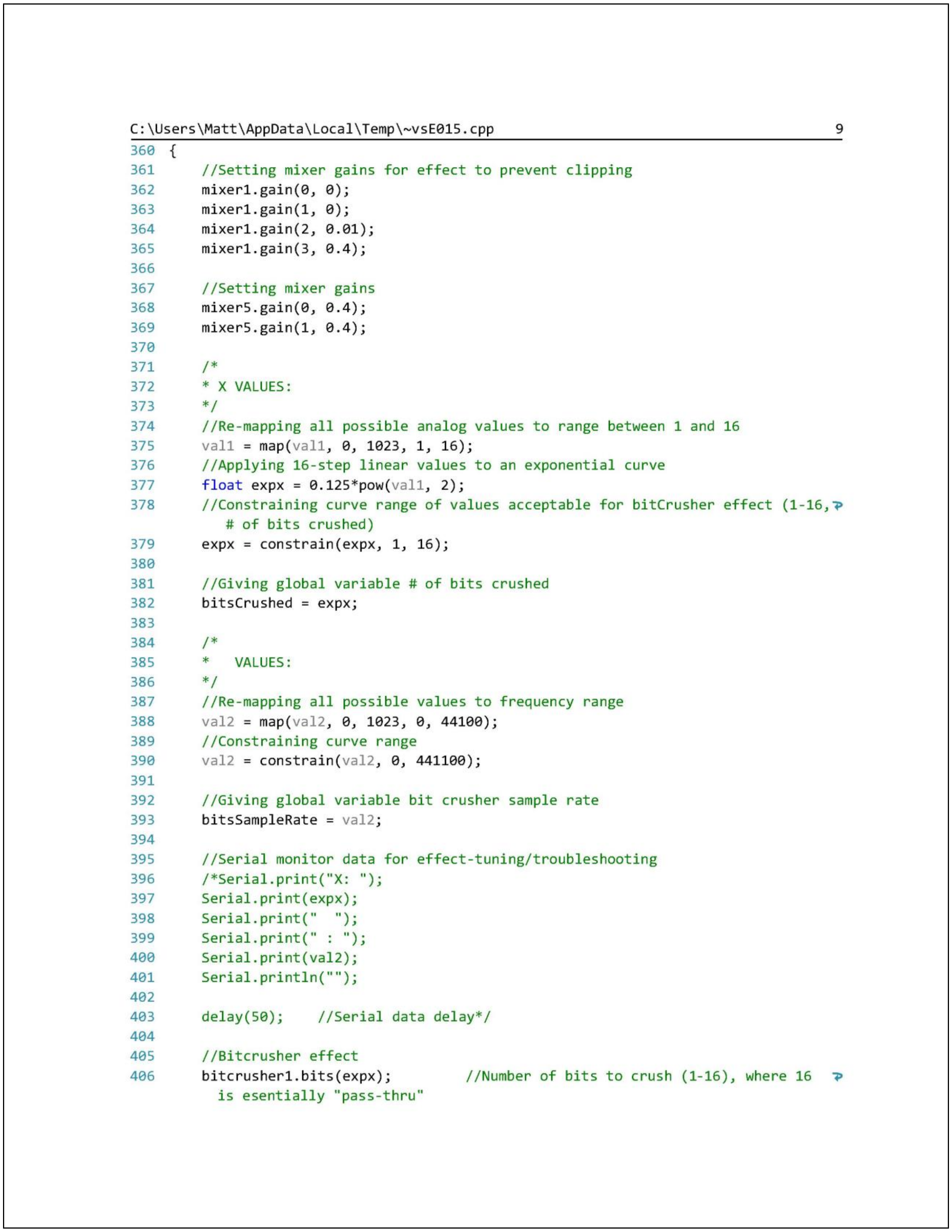
C:\Users\Matt\AppData\Local\Temp\~vsE015.cpp 4

```
//Declare fucntion
int avgDelayTime(int joyPosition);


SSD1306AsciiWire oled;
// 0X3C+SA0 - 0x3C or 0x3D
#define I2C_ADDRESS 0x3C

void setup() {
    Serial.begin(9600);   //Begin serial

    Wire.begin();                               //Start OLED communucation
    oled.begin(&Adafruit128x32, I2C_ADDRESS);   //Assign OLED i2c address
    oled.setFont(Arial14);                      //Selecting OLED font style

                                                //Defining selector switch to internally pull-up
    pinMode(pos1, INPUT_PULLUP);
    pinMode(pos2, INPUT_PULLUP);

    //Defining joystick pushbutton switch to internally pull-up
    pinMode(btn1, INPUT_PULLUP);

    //Setting inital audio-control chip parameters
    sgtl5000_1.enable();      //Enable audio chip control
    sgtl5000_1.volume(0.4);   //Preset I2S output volume
    AudioMemory(200);         //Allocate ram
}





void loop() {
    //Read selector switch position
    selectTop = digitalRead(pos1);
    selectBtm = digitalRead(pos2);

    //Read joystick position
    xpos = analogRead(vrx);
    ypos = analogRead(vry);

    //Read volume pot
    volRaw = analogRead(volPotPin);


    /*Read joystick pushbutton switch
    * inital values: currentPress == 0, lastPress == 1
    */
```

```cpp
    }

    //Determining desired audio effect
    if (selectTop == LOW)    //REVERB EFFECT
    {
        if (lastEffect  = 1)
        {
            oled.clear();
            oled.setFont(Arial_bold_14);
            oled.print("Reverb");
            oled.setFont(Arial14);

            oled.setCursor(65, 2);
            oled.print("ms");

            oled.setCursor(0, 2);
            oled.print("Delay: ");
        }

        //Displaying Effect Parameters:
        oled.setCursor(40, 2);
        oled.clear(40, 63, 2, 4);
        oled.print(avgDelayTime(xpos));

        reverb(xpos);
        lastEffect = 1;
    }
    else if (selectBtm == LOW)   //DISTORTION EFFECT
    {
        if (lastEffect  = 2)
        {
            oled.setFont(Arial_bold_14);
            oled.clear();
            oled.print("Distortion");
            oled.setFont(Arial14);
        }

        //Displaying Effect Parameters for # of bits crushed:
        oled.setCursor(0, 2);
        oled.print("Bits: ");
        oled.print(bitsCrushed);
        oled.print("  ");   //Printing blank spaces to clear "stuck pixels"

                        //Displaying Effect Parameters for Bit Crusher Sample Rate:
        oled.setCursor(40, 2);
        oled.print(" Rate: ");
        oled.print((float)bitsSampleRate / 1000);
        oled.print("   ");    //Printing blank spaces to clear "stuck pixels"
```

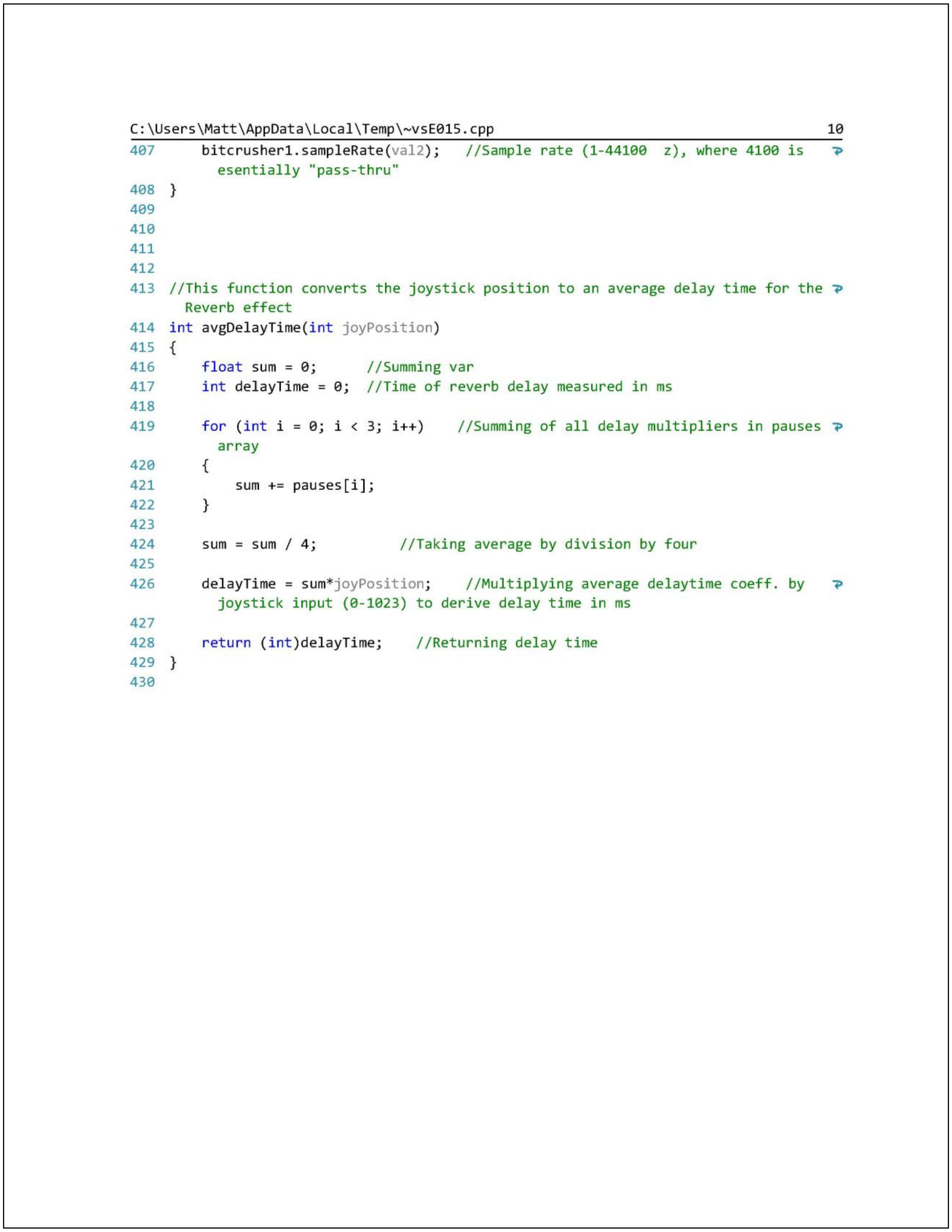

C:\Users\Matt\AppData\Local\Temp\~vsE015.cpp 7

```cpp
        oled.setCursor(105, 2);
        oled.print(" k z");


        distortion(xpos, ypos);
        lastEffect = 2;
    }
    else    //PASS-T RU AUDIO
    {
        if (lastEffect  = 3)
        {
            oled.setFont(Arial_bold_14);
            oled.clear();
            oled.print("No Effect");
            oled.setFont(Arial14);
        }
        passThru();
        lastEffect = 3;
    }





}




//Pass-through (no effect) function
void passThru()
{
    mixer1.gain(0, 0);
    mixer1.gain(1, 0);
    mixer1.gain(2, 0);
    mixer1.gain(3, 0.7);
}



//Reverb effect
void reverb(int val1)
{
    //Setting final-stage mixer gains
    mixer1.gain(0, 0.6);
    mixer1.gain(1, 0.4);
    mixer1.gain(2, 0);
    mixer1.gain(3, 0);

    //Setting feedback mixer gains (C .3 unused)
    mixer2.gain(0, 0.4);    //attenuate pure signal to prevent clipping
    mixer2.gain(1, 0.4);    //attenuate pure signal to prevent clipping
    mixer2.gain(2, 0.25);   //attentuate feedback gain

                            //Setting delay (mixer3) gains to prevent clipping
    mixer3.gain(0, 0.2);
    mixer3.gain(1, 0.2);
    mixer3.gain(2, 0.2);
    mixer3.gain(3, 0.2);

    //Setting delay (mixer4) gains to prevent clipping
    mixer4.gain(0, 0.2);
    mixer4.gain(1, 0.2);
    mixer4.gain(2, 0.2);
    mixer4.gain(3, 0.2);

    //Lowpass filter to reduce signal noise
    biquad1.setLowpass(3, 4000, 0.699);

    //Set final-stage mixer such that reverb-effect is "OFF" when joystick value is less than 50.
    if (val1 < 50)
    {
        mixer1.gain(1, 0);    //If delay value less than 50, turn off delay completely to reduce noticable clipping and noise resulting from delay-line
    }

    //Applying pauses[] coefficents to delaylines
    for (int i = 0; i < 8; i++)
    {
        if (i < 4)
        {
            delay1.delay(i, val1*pauses[i]);    //Setting delay durations for first "four" (of eight) channels (0-3)
        }
        else if (i >= 4)
        {
            delay1.delay(i, val1*pauses[i - 4]);    //Setting delay durations for second "four" (of eight) channel (4-7)
        }
    }
}



//Distortion efect
void distortion(int val1, int val2)
```



C:\Users\Matt\AppData\Local\Temp\~vsE015.cpp 9

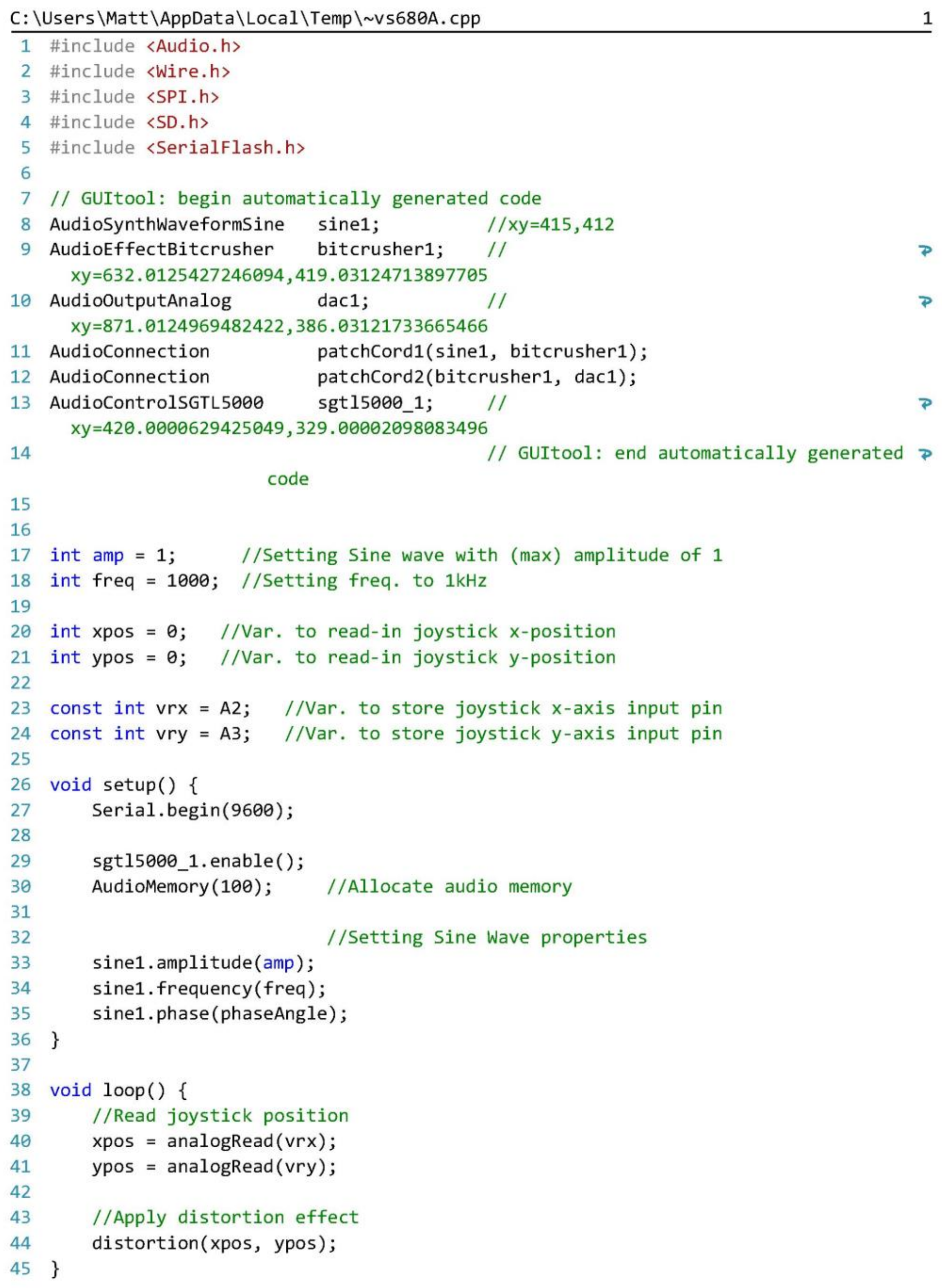
```cpp
{
    //Setting mixer gains for effect to prevent clipping
    mixer1.gain(0, 0);
    mixer1.gain(1, 0);
    mixer1.gain(2, 0.01);
    mixer1.gain(3, 0.4);

    //Setting mixer gains
    mixer5.gain(0, 0.4);
    mixer5.gain(1, 0.4);

    /*
    * X VALUES:
    */
    //Re-mapping all possible analog values to range between 1 and 16
    val1 = map(val1, 0, 1023, 1, 16);
    //Applying 16-step linear values to an exponential curve
    float expx = 0.125*pow(val1, 2);
    //Constraining curve range of values acceptable for bitCrusher effect (1-16, # of bits crushed)
    expx = constrain(expx, 1, 16);

    //Giving global variable # of bits crushed
    bitsCrushed = expx;

    /*
    *   VALUES:
    */
    //Re-mapping all possible values to frequency range
    val2 = map(val2, 0, 1023, 0, 44100);
    //Constraining curve range
    val2 = constrain(val2, 0, 441100);

    //Giving global variable bit crusher sample rate
    bitsSampleRate = val2;

    //Serial monitor data for effect-tuning/troubleshooting
    /*Serial.print("X: ");
    Serial.print(expx);
    Serial.print("  ");
    Serial.print(" : ");
    Serial.print(val2);
    Serial.println("");

    delay(50);    //Serial data delay*/

    //Bitcrusher effect
    bitcrusher1.bits(expx);         //Number of bits to crush (1-16), where 16 is esentially "pass-thru"
```



C:\Users\Matt\AppData\Local\Temp\~vsE015.cpp 10

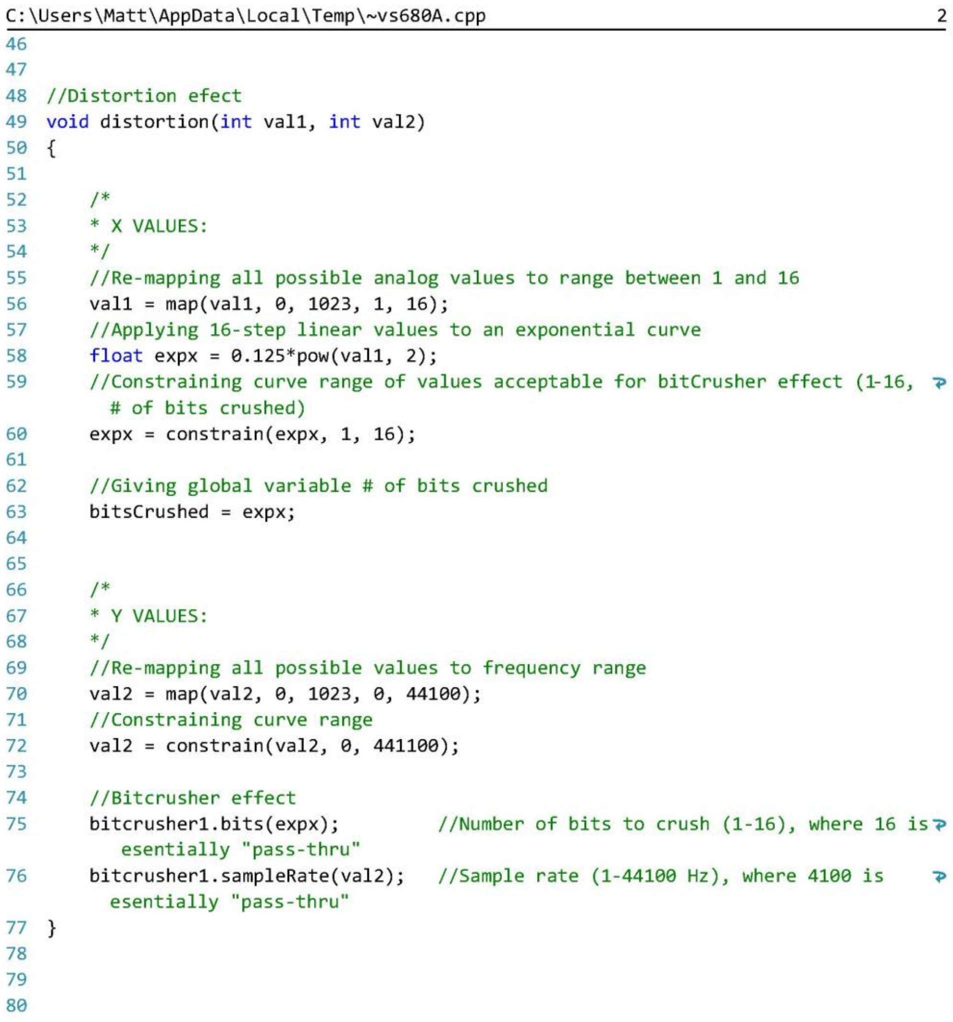
```cpp
    bitcrusher1.sampleRate(val2);   //Sample rate (1-44100  z), where 4100 is esentially "pass-thru"
}




//This function converts the joystick position to an average delay time for the Reverb effect
int avgDelayTime(int joyPosition)
{
    float sum = 0;      //Summing var
    int delayTime = 0;  //Time of reverb delay measured in ms

    for (int i = 0; i < 3; i++)    //Summing of all delay multipliers in pauses array
    {
        sum += pauses[i];
    }

    sum = sum / 4;          //Taking average by division by four

    delayTime = sum*joyPosition;    //Multiplying average delaytime coeff. by joystick input (0-1023) to derive delay time in ms

    return (int)delayTime;    //Returning delay time
}

```

[Attachment 2] Distortion Audio Test Sketch



C:\Users\Matt\AppData\Local\Temp\~vs680A.cpp 1

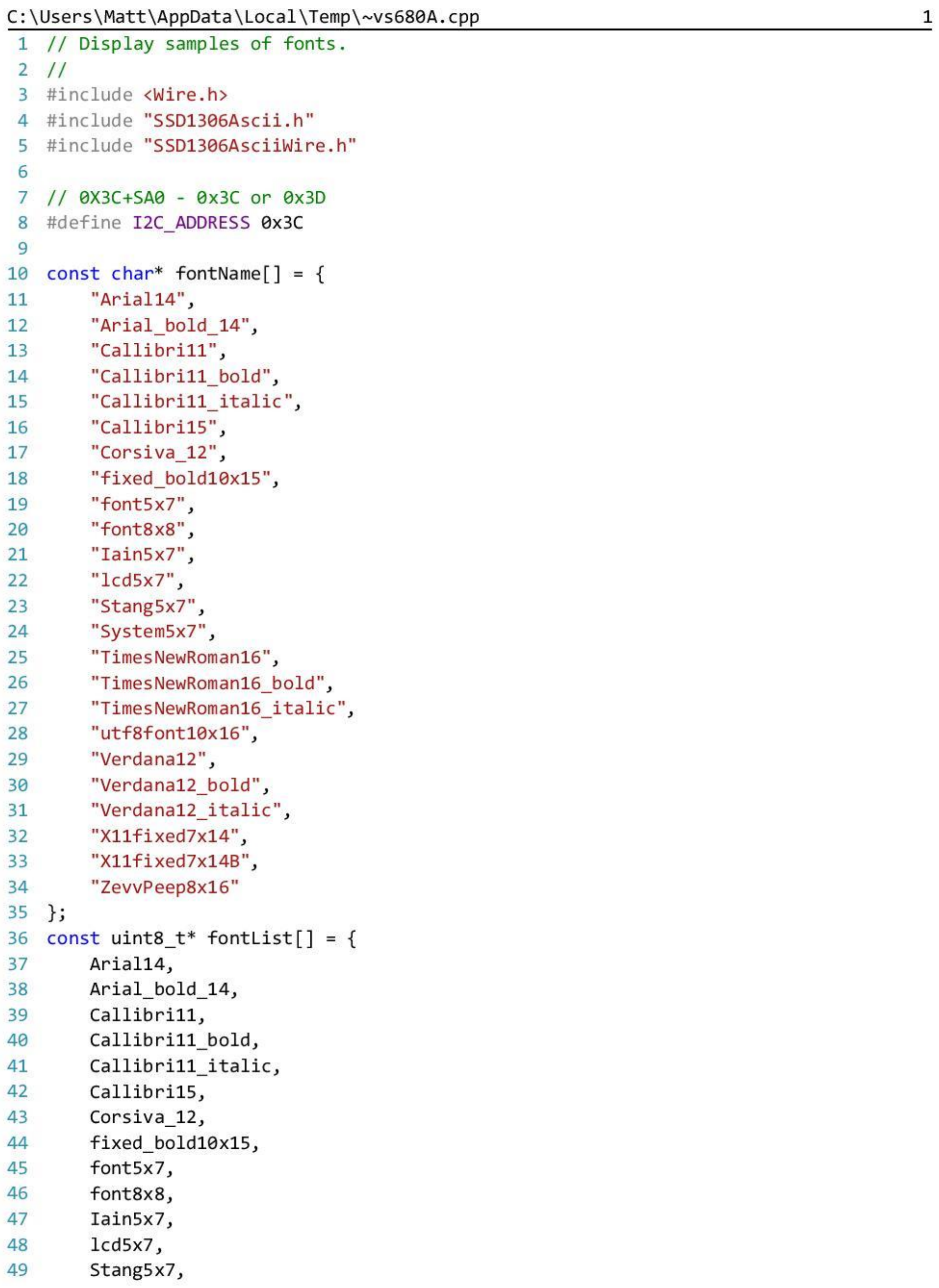
```cpp
#include <Audio.h>
#include <Wire.h>
#include <SPI.h>
#include <SD.h>
#include <SerialFlash.h>

// GUItool: begin automatically generated code
AudioSynthWaveformSine   sine1;          //xy=415,412
AudioEffectBitcrusher    bitcrusher1;    //xy=632.0125427246094,419.03124713897705
AudioOutputAnalog        dac1;           //xy=871.0124969482422,386.03121733665466
AudioConnection          patchCord1(sine1, bitcrusher1);
AudioConnection          patchCord2(bitcrusher1, dac1);
AudioControlSGTL5000     sgtl5000_1;     //xy=420.0000629425049,329.00002098083496
                                         // GUItool: end automatically generated code


int amp = 1;      //Setting Sine wave with (max) amplitude of 1
int freq = 1000;  //Setting freq. to 1kHz

int xpos = 0;   //Var. to read-in joystick x-position
int ypos = 0;   //Var. to read-in joystick y-position

const int vrx = A2;   //Var. to store joystick x-axis input pin
const int vry = A3;   //Var. to store joystick y-axis input pin

void setup() {
    Serial.begin(9600);

    sgtl5000_1.enable();
    AudioMemory(100);     //Allocate audio memory

                          //Setting Sine Wave properties
    sine1.amplitude(amp);
    sine1.frequency(freq);
    sine1.phase(phaseAngle);
}

void loop() {
    //Read joystick position
    xpos = analogRead(vrx);
    ypos = analogRead(vry);

    //Apply distortion effect
    distortion(xpos, ypos);
}
```



C:\Users\Matt\AppData\Local\Temp\~vs680A.cpp 2

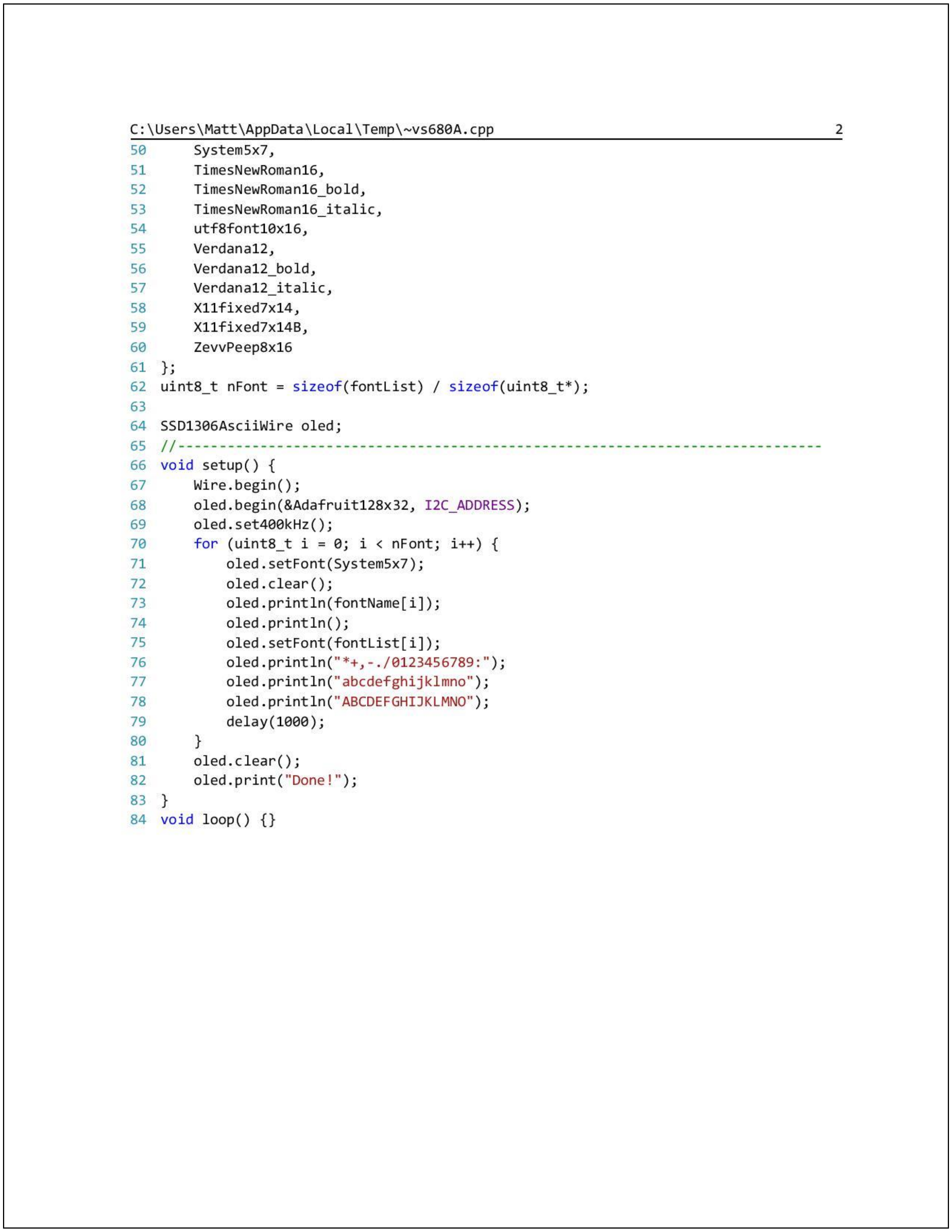
```cpp


//Distortion efect
void distortion(int val1, int val2)
{

    /*
    * X VALUES:
    */
    //Re-mapping all possible analog values to range between 1 and 16
    val1 = map(val1, 0, 1023, 1, 16);
    //Applying 16-step linear values to an exponential curve
    float expx = 0.125*pow(val1, 2);
    //Constraining curve range of values acceptable for bitCrusher effect (1-16, # of bits crushed)
    expx = constrain(expx, 1, 16);

    //Giving global variable # of bits crushed
    bitsCrushed = expx;


    /*
    * Y VALUES:
    */
    //Re-mapping all possible values to frequency range
    val2 = map(val2, 0, 1023, 0, 44100);
    //Constraining curve range
    val2 = constrain(val2, 0, 441100);

    //Bitcrusher effect
    bitcrusher1.bits(expx);          //Number of bits to crush (1-16), where 16 is esentially "pass-thru"
    bitcrusher1.sampleRate(val2);    //Sample rate (1-44100 Hz), where 4100 is esentially "pass-thru"
}



```

[Attachment 3] SSD1306 (Greiman ASCII Library) OLED Test Sketch



C:\Users\Matt\AppData\Local\Temp\~vs680A.cpp 1

```
// Display samples of fonts.
//
#include <Wire.h>
#include "SSD1306Ascii.h"
#include "SSD1306AsciiWire.h"

// 0X3C+SA0 - 0x3C or 0x3D
#define I2C_ADDRESS 0x3C

const char* fontName[] = {
    "Arial14",
    "Arial_bold_14",
    "Callibri11",
    "Callibri11_bold",
    "Callibri11_italic",
    "Callibri15",
    "Corsiva_12",
    "fixed_bold10x15",
    "font5x7",
    "font8x8",
    "Iain5x7",
    "lcd5x7",
    "Stang5x7",
    "System5x7",
    "TimesNewRoman16",
    "TimesNewRoman16_bold",
    "TimesNewRoman16_italic",
    "utf8font10x16",
    "Verdana12",
    "Verdana12_bold",
    "Verdana12_italic",
    "X11fixed7x14",
    "X11fixed7x14B",
    "ZevvPeep8x16"
};
const uint8_t* fontList[] = {
    Arial14,
    Arial_bold_14,
    Callibri11,
    Callibri11_bold,
    Callibri11_italic,
    Callibri15,
    Corsiva_12,
    fixed_bold10x15,
    font5x7,
    font8x8,
    Iain5x7,
    lcd5x7,
    Stang5x7,
```



C:\Users\Matt\AppData\Local\Temp\~vs680A.cpp 2

```cpp
    System5x7,
    TimesNewRoman16,
    TimesNewRoman16_bold,
    TimesNewRoman16_italic,
    utf8font10x16,
    Verdana12,
    Verdana12_bold,
    Verdana12_italic,
    X11fixed7x14,
    X11fixed7x14B,
    ZevvPeep8x16
};
uint8_t nFont = sizeof(fontList) / sizeof(uint8_t*);

SSD1306AsciiWire oled;
//------------------------------------------------------------------------------
void setup() {
    Wire.begin();
    oled.begin(&Adafruit128x32, I2C_ADDRESS);
    oled.set400kHz();
    for (uint8_t i = 0; i < nFont; i++) {
        oled.setFont(System5x7);
        oled.clear();
        oled.println(fontName[i]);
        oled.println();
        oled.setFont(fontList[i]);
        oled.println("*+,-./0123456789:");
        oled.println("abcdefghijklmno");
        oled.println("ABCDEFGHIJKLMNO");
        delay(1000);
    }
    oled.clear();
    oled.print("Done!");
}
void loop() {}
```

10. References

[1] PJRC.com, "Electronic Projects Components Worldwide," [Online]. Available: https://pjrc.com. [Accessed: 8 Dec., 2017].

[2] Amazon.com, "Teensy 3.2," [Online]. Available: https://www.amazon.in/6485230-Teensy-3-2/dp/B015M3K5NG. [Accessed: 8 Dec., 2017].

[3] Sparkfun.com, "3.3V CMOS Logic Levels," [Online]. Available: https://learn.sparkfun.com/tutorials/logic-levels#33-v-cmos-logic-levels. [Accessed: 8 Dec., 2017].

[4] PJRC.com, "Teensy 3.2 and 3.1 – New Features," [Online]. Available: https://www.pjrc.com/teensy/teensy31.html. [Accessed: 8 Dec., 2017].

[5] ARM.com, "Cortex-M4 Processor," [Online]. Available: https://www.arm.com/products/processors/cortex-m4-processor.php. [Accessed: 8 Dec., 2017].

[6] PJRC.com, "Teensyduino," [Online]. Available: https://www.pjrc.com/teensy/teensyduino.html. [Accessed: 8 Dec., 2017].

[7] PJRC.com, "Audio Adaptor Board for Teensy 3.0 – 3.6," [Online]. Available: https://www.pjrc.com/store/teensy3_audio.html. [Accessed: 8 Dec., 2017].

[8] Measurement Computing, "Analog to Digital Conversion," [Online]. Available: https://www.mccdaq.com/PDFs/specs/Analog-to-Digital.pdf. [Accessed: 8 Dec., 2017].

[9] Wikipedia.com, "Digital signal processor," [Online]. Available: https://en.wikipedia.org/wiki/Digital_signal_processor. [Accessed: 8 Dec., 2017].

[10] NXP, "Low Power Stereo Codec with Headphone Amp," SGTL5000 datasheet, Nov. 11 2017 [Online]. Available: https://www.pjrc.com/teensy/SGTL5000.pdf. [Accessed: 8 Dec., 2017].

[11] Philips Semiconductors, "I^{2}S bus specification," [Online]. Available: https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf. [Accessed: 8 Dec., 2017].

[12] PubNub, "How Fast is Realtime? Human Perception and Technology," [Online]. Available: https://www.pubnub.com/blog/2015-02-09-how-fast-is-realtime-human-perception-and-technology/. [Accessed: 8 Dec., 2017].

[13] Presonus.com, “Digital Audio Latency Explained,” [Online]. Available: https://www.presonus.com/learn/technical-articles/Digital-Audio-Latency-Explained. [Accessed: 8 Dec., 2017].

[14] Stanford.edu, “Freeverb,” [Online]. Available: https://ccrma.stanford.edu/~jos/pasp/Freeverb.html. [Accessed: 8 Dec., 2017].

[15] DesigningSound.org, J.Menhorn “Reverb: The Science And The State-of-the-Art,” [Online]. Available: http://designingsound.org/2012/12/reverb-the-science-and-the-state-of-the-art/. [Accessed: 8 Dec., 2017].

[16] GMArts.com, “Overdrive & Distortion,” [Online]. Available: http://www.gmarts.org/index.php?go=217. [Accessed: 8 Dec., 2017].

[17] Sound.Whsites.net, “Soft Clipping,” [Online]. Available: http://sound.whsites.net/articles/soft-clip.htm. [Accessed: 8 Dec., 2017].

[18] PJRC.com, “Teensy Audio Library,” [Online]. Available: http://www.pjrc.com/teensy/td_libs_Audio.html. [Accessed: 8 Dec., 2017].

[19] Audiocheck.net, “The non-linearities of the Human Ear,” [Online]. Available: http://www.audiocheck.net/soundtests_nonlinear.php. [Accessed: 8 Dec., 2017].

[20] Brilliant.org, “Finite State Machines,” [Online]. Available: https://brilliant.org/wiki/finite-state-machines/. [Accessed: 8 Dec., 2017].

[21] Earlevel.com, “Biquads,” [Online]. Available: http://www.earlevel.com/main/2003/02/28/biquads/. [Accessed: 8 Dec., 2017].

[22] PJRC.com, “Bounce Library,” [Online]. Available: https://www.pjrc.com/teensy/td_libs_Bounce.html. [Accessed: 8 Dec., 2017].

[23] Github.com, “Audio.h Library,” [Online]. Available: https://github.com/PaulStoffregen/Audio/blob/master/Audio.h. [Accessed: 8 Dec., 2017].

[24] Arduino.cc, “Wire Library,” [Online]. Available: https://www.arduino.cc/en/Reference/Wire. [Accessed: 8 Dec., 2017].

[25] Arduino.cc, “SPI Library,” [Online]. Available: https://www.arduino.cc/en/Reference/SPI. [Accessed: 8 Dec., 2017].

[26] Arduino.cc, “SD Library,” [Online]. Available: https://www.arduino.cc/en/Reference/SD. [Accessed: 8 Dec., 2017].

[27] Github.com, “Electronic Projects Components Worldwide,” [Online]. Available: https://github.com/PaulStoffregen/SerialFlash. [Accessed: 8 Dec., 2017].

[28] Github.com, “Electronic Projects Components Worldwide,” [Online]. Available: https://github.com/greiman/SSD1306Ascii. [Accessed: 8 Dec., 2017].

[29] CCSinfo.com, “I2C Driver for SSD1306 Graphic Chip,” [Online]. Available: https://www.ccsinfo.com/forum/viewtopic.php?t=52836. [Accessed: 8 Dec., 2017].

[30] Arduino.cc, “Input Pullup Serial,” [Online]. Available: https://www.arduino.cc/en/Tutorial/InputPullupSerial. [Accessed: 8 Dec., 2017].

[31] PJRC.com, “Audio Connections and Memory,” [Online]. Available: https://www.pjrc.com/teensy/td_libs_AudioConnection.html. [Accessed: 8 Dec., 2017].

[32] Arduino.cc, “Map Function,” [Online]. Available: https://www.arduino.cc/reference/en/language/functions/math/map/. [Accessed: 8 Dec., 2017].

[33] M. Wax, “The estimate of time delay between two signals with random relative phase shift,” In IEEE International Conference on ICASSP ’81, 1981.