Audio Effects Project

ECE 299

July 29, 2018

Robert Lee
Declan McIntosh

This page was intentionally left blank.

Acknowledgment

We would like to thank several people that have significantly helped during the progress of this project. We would like to acknowledge Brent, the laboratory technician at the University of Victoria, for his help detailing the specific implementation of the Fast Fourier Transform used in this project and for writing the FFT code. We would like to also thank Brent for helping confirm the validity of the deigned circuits before the submission of these circuits to the manufacturer.

We would like to thank Dr. T. Ilamparithi for his guidance and support, and for his lectures which gave us the theoretical knowledge required for this project.

We would like to acknowledge the help provided by our tutorial instructor Alireza Rahimpour during the experiments that taught the tools required for this project.

Further recognition of the manufacturer used, jlcpcb.com, for manufacturing a high quality and effective printed circuit board in a short period of time, and to DHL for their strong logistics network to move the ordered PCBs from Shenzhen, China to Victoria, BC, Canada remarkable quickly.

Finally, the writers of this report would like to extend our appreciation to our friends and family who have always supported us.

Contents

Acknowledgment i
Contents ii
List of Tables iii
List of Figures iv
I. Project Goal 1
II. Constraints 1
III. Requirement Specifications 1
IV. Bill of Materials 4
V. Circuit Schematic & PCB 8
VI. Testing & Validation 15
VII. Conclusion & Recommendations 20
References 22
Appendix A – Enclosure 23
Appendix B – Gantt Chart for Hypothetical Projects 27
Implementation using the STM32F4 Discovery Board 27
Implementation using the TMS320F28035 Piccolo development board 27
Appendix C – Sample Code 31

List of Tables

Table 1: Bill of Materials [6] 5
Table 2: Component Choice Justification.. 7

List of Figures

Figure 1: Voltage Reference Schematic 2
Figure 2: DAC Quantization Error Amelioration Circuit Schematic 3
Figure 3: Calculator results for a cutoff frequency of ~10.4 kHz...................................... 3
Figure 4: LM386 Schematic 4
Figure 5: Audio FX: Logic and Audio Signal Processing Board 9
Figure 6: Audio FX: LED Array Board 10
Figure 7: Logic and Audio Signal Processing Board PCB Layout.................................. 11
Figure 8: Audio FX: LED Array Board PCB Layout 12
Figure 9: Logic and Audio Signal Processing Board Bare PCB 13
Figure 10: Audio FX: LED Array Board Bare PCB 13
Figure 11: Logic and Audio Signal Processing Board Populated PCB 14
Figure 12: LED Array Board Populated PCB.. 15
Figure 13: Oscilloscope display of the audio amplifier scaling the input signal by approximately 20 times. Yellow corresponds to output and green corresponds to input. 19
Figure 14: Oscilloscope display showing the attenuation of a signal past the cut-off frequency of 10.4kHz. Green corresponds to input and yellow corresponds to output on the same scale. 20
Figure 15: Enclosure for the Audio Effects board and LED display. Note the agronomical button placement on the face of the product. 23
Figure 16: The rear of the enclosure, showing the mounting hole locations for the main circuit PCB. 24
Figure 17: Multiple views of the enclosure designed using SolidWorks a....................... 25
Figure 18: Gantt chart for first hypothetical scenario: design using STM32F4 Discovery Board 29
Figure 19: Gantt chart for second hypothetical scenario: design using TMS320F28035 Piccolo development board 29

I. Project Goal

The goal of this project was to create an audio player which can take an analog input and perform some transformations (pitch and echo) on the music, then output this modified music with an imbedded speaker. This unit should have a convenient user interface and a vibrant display.

II. Constraints

Several constraints were placed on the project by the customer. The project required the use of the STM32F4 Discovery board (STM32 board) for digital signal processing (DSP) and logical processing. The board and all peripherals on the printed circuit board (PCB) must be powered solely by a USB port on a computer. USB ports will provide 5 V at 0.1 A for low power devices before handshaking negotiations are required [1]. The STM32 board must be used to implement pitch shifting and add an echo effect. The software must be written in the Eclipse Integrated Development Environment (IDE). The PCB must be designed using KiCAD. The designed enclosure (see **Appendix A**) must be designed in SolidWorks. The PCB and all components must be RoHS (Restriction of Hazardous Substances) compliant. A cost constraint of $150 CAD was decided by the group.

III. Requirement Specifications

The product must display the current effect which is added to the audio. The minimum requirement is through the use of a four-digit seven-segment LED display. However, to facilitate enhanced customizability, an eight-by-eight LED matrix was used. The display was drawn one pixel at a time, progressively down each column, then incrementing up the columns. The chosen refresh rate for the entire display was 250 Hz, the minimum frequency experimentally found to eliminate flickering. Because there are 64 LEDs that need to be individually lit for one refresh of the entire display, the timer frequency driving the interrupt service routine (ISR) was chosen to be

$$250\,\frac{full\ display\ refresh}{second} \times 64\,\frac{LEDs}{full\ display\ refresh} = 16\ kHz \quad (1)$$

The refresh rate for the entire display was increased to 5000 Hz to move the noise that was introduced to the speaker into the inaudible frequency range. Thus, the timer frequency was changed to

$$5000\,\frac{full\ display\ refresh}{second} \times 64\,\frac{LEDs}{full\ display\ refresh} = 320\ kHz \quad (2)$$

To achieve this, the prescaler was chosen to be 105, and the timer period was chosen to be 2, so that, for *TIMER4*'s maximum clock of 84 MHz, the *timer tick frequency* and *timer frequency* can be calculated to be [2]

$$TimerTickFrequency = \frac{Max\ clock\ frequency}{Prescaler+1} = \frac{84\ 000\ 000\ Hz}{104+1} = 800\ 000\ Hz \quad (3)$$

$$TimerFrequency = \frac{TimerTickFrequency}{Period+1} = \frac{800\,000\ Hz}{1+1} = 400\ kHz \quad (4)$$

which is near the desired frequency.

Button debouncing was implemented in software using another timer, *TIMER3*, at 200 Hz driving an ISR that reads the button states. The ISR stores the previous state of the button in memory and compares it to the current state. If the two states are consistent across readings, the ISR updates a global button state variable that fully encapsulates the button debouncing, so it can be reliably used in other functions. If pitch shifting is enabled, this ISR also reads the 8-bit potentiometer voltage value and writes this to a global variable. Using similar calculations to the equations shown above, the prescaler was chosen to be 49, and the period was chosen to be 8399, to give a timer frequency of 200 Hz.

There were three major analog circuits within the PCB:

- *Level shifter circuit*: since the input signal is sinusoidal centered at 0 V, and the Analog-to-Digital Converter (ADC) accepts a voltage range from 0 V to 3 V, the input signal must be shifted up.
- *Digital-to-Analog Converter (DAC) Quantization Error Amelioration Circuit*: since the DAC can only output discrete voltages, not a continuous range of voltages, there are a lot of jagged bumps in the signal. These arise as high-frequency noise in the signal and are removed with a lowpass filter.
- *Audio amplification and bandpass filtering prior to output to speaker*: the LM386 amplifier was operated with a gain of 20 [3]. The bandpass filter consists of a DC blocking capacitor and a lowpass filter.

The level shifter circuit is comprised of a voltage reference connected to a voltage follower. This is then used to voltage divide the input signal using an operational amplifier. The following equation in the datasheet [4] and the following schematic (see Figure 1) were used,

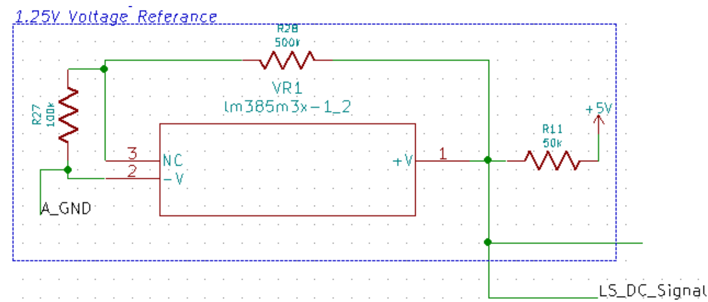

Figure 1: Voltage Reference Schematic

$$V_{LS_DC_Signal} = 1.24(\frac{R_{27}}{R_{28}} + 1) \quad (5)$$

where $V_{LS_DC_Signal}$ is the voltage reference value (volts), and R_{27} and R_{28} are resistors (ohms). R_{27} was set to 100 kΩ and R_{28} was set to 500 kΩ, so the reference voltage was 1.488 V.

The DAC quantization error amelioration circuit was implemented using a second-order active lowpass filter. The following schematic (see Figure 2) and the online calculator [5] (see Figure 3) were used:

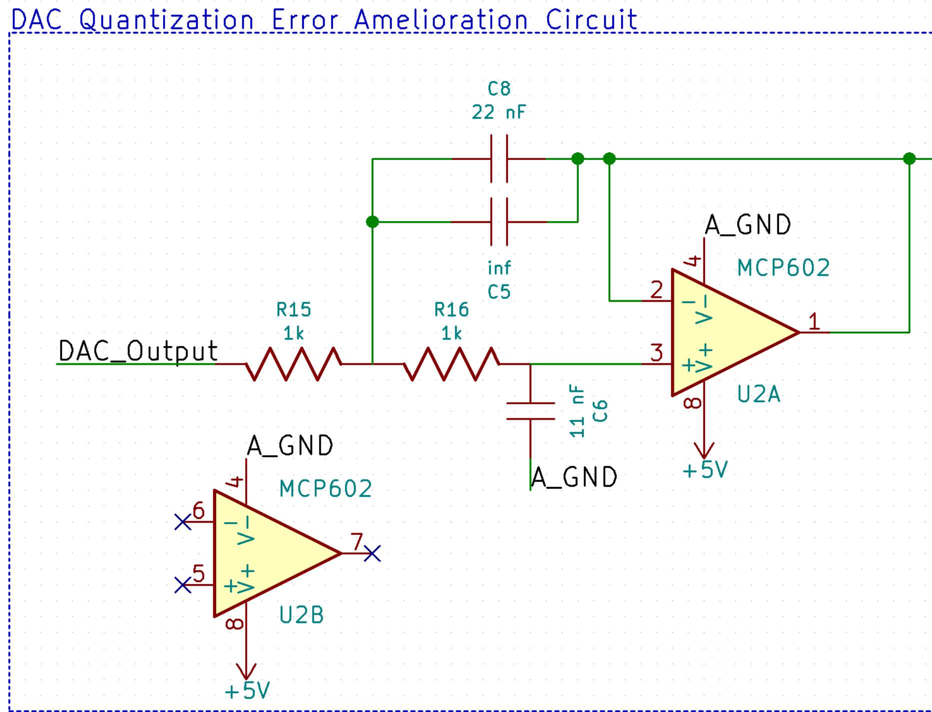

Figure 2: DAC Quantization Error Amelioration Circuit Schematic

$$C_a = \frac{\text{Sqrt (2)}}{2 \times 3.14 \times \text{Cutoff Frequency} \times \text{Resistor}}$$

$$C_b = \frac{C_a}{2}$$

Enter your values:

Cutoff Frequency: 10.4 KHz

Results:
(Enter any one value - Resistor or C_a or C_b)

Resistor (R_a = R_b): 1.000000 Kohm
Capacitor C_a: 21.63462 nF
Capacitor C_b: 10.81731 nF

Figure 3: Calculator results for a cutoff frequency of ~10.4 kHz

where C_a is capacitor C_6, C_b is capacitor C_8, R_a and R_b are resistors R_{15} and R_{16}. The calculated cutoff frequency was approximately 10.4 kHz. The ideal cutoff frequency would be 8 kHz, but limitations were found in the available capacitor and resistor sizes.

The LM386 audio amplifier was operated with a gain of 20 when the following circuit is used (see Figure 4).

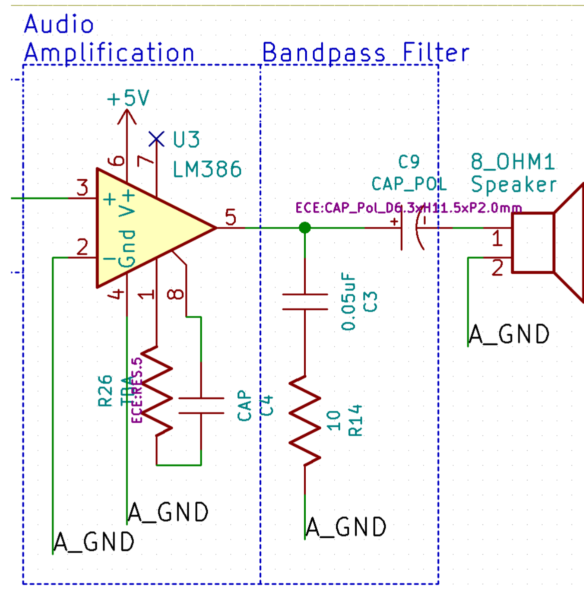

Figure 4: LM386 Schematic

where R26 and CAP were left empty. This circuit was the suggested circuit on the datasheet for a gain of 20 [3].

The code is broken down into the following major components, and is supplied as **Appendix C**:

- Timer-run interrupt service routine to drive the LED 8-by-8 matrix display, as described above
- Timer-run interrupt service routine to poll input buttons and pitch shifting potentiometer, as described above
- Timer-run interrupt service routine to read ADC audio input, and to output processed data using the DAC
- Finite State Machine logic to determine the current state, and to display it on the LED matrix display
- LED matrix display backend functions and data, to enable the user to fill the display buffer with up to 10 seconds of images, played at 25 frames per second

IV. Bill of Materials

The following tables detail all the required materials for the implemented solution excluding the PCB itself and the enclosure.

Table 1: Bill of Materials [6]

Label in schematic	Component Description	Part number	Cost/unit quantity (CAD)	Source of cost information
8_Ohm1	Speaker	CDMG15008-03A-ND	3.13	DigiKey
C1	$0.05\mu F$ Capacitor	BC2686CT-ND	0.28	DigiKey
C3	$0.05\mu F$ Capacitor	BC2686CT-ND	0.28	DigiKey
C4	Unused Capacitor	Unused	Unused	Unused
C6	$11nF$ Capacitor	SA105C143JAR-ND	0.19	DigiKey
C7	$220\mu F$ Capacitor	1189-1546-3-ND	0.39	DigiKey
C8	$22nF$ Capacitor	SA105C143JAR-ND	0.19	DigiKey
C9	$220\mu F$ Capacitor	1189-1546-3-ND	0.39	DigiKey
P1	100k Potentiometer	P160KNP-0EC15A100K	1.06	DigiKey
P2	100k Potentiometer	P160KNP-0EC15A100K	1.06	DigiKey
P3	100k Potentiometer	P160KNP-0EC15A100K	1.06	DigiKey
P4	100k Potentiometer	P160KNP-0EC15A100K	1.06	DigiKey
P5	100k Potentiometer	P160KNP-0EC15A100K	1.06	DigiKey
R1	100k Ohm 1/16W Resistor	CF14JT100KCT-ND	0.15	DigiKey
R2	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R3	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R4	100k Ohm 1/16W Resistor	CF14JT100KCT-ND	0.15	DigiKey
R5	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R6	100k Ohm 1/16W Resistor	CF14JT100KCT-ND	0.15	DigiKey
R7	100k Ohm 1/16W Resistor	CF14JT100KCT-ND	0.15	DigiKey
R8	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R9	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R10	100k Ohm 1/16W Resistor	CF14JT100KCT-ND	0.15	DigiKey
R11	50k Ohm 1/16W Resistor	CF18JT150KCT-ND	0.15	DigiKey
R12	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey

R13	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R14	10 OHM 1W 5% AXIAL	FW10A10R0JACT-ND	0.81	DigiKey
R15	1k Ohm 1/16W Resistor	CF14JT1K00CT-ND	0.15	DigiKey
R16	1k Ohm 1/16W Resistor	CF14JT1K00CT-ND	0.15	DigiKey
R17	10 Ohm 1W Resistor	FW10A10R0JACT-ND	0.81	DigiKey
R18	100k Ohm 1/16W Resistor	CF14JT100KCT-ND	0.15	DigiKey
R19	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R20	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R21	0 Ohm 3W Resistor	JW50ZT0R00CT-ND	0.15	DigiKey
R22	0 Ohm 3W Resistor	JW50ZT0R00CT-ND	0.15	DigiKey
R23	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R24	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R25	10k Ohm 1/16W Resistor	CF14JT10K0CT-ND	0.15	DigiKey
R27	100k Ohm 1/16W Resistor	CF14JT100KCT-ND	0.15	DigiKey
R28	500k Ohm 1/16W Resistor	500KAACT-ND	0.44	DigiKey
STM1	STM32F4 Discovery Board	497-15211-ND	21.07	DigiKey
SW1	Push Button Switch	PS1024ALBLK	1.71	DigiKey
SW2	Push Button Switch	PS1024ALBLK	1.71	DigiKey
SW3	Push Button Switch	PS1024ALBLK	1.71	DigiKey
U1	MCP602 Rail to Rail Op Amp	MCP602-I/P-ND	0.88	DigiKey
U2	MCP602 Rail to Rail Op Amp	MCP602-I/P-ND	0.88	DigiKey
U3	LM386 Audio Amplifier	296-43960-5-ND	1.51	DigiKey
VR1	LM385 Voltage Reference	LM385BZ-2.5GOS-ND	0.97	DigiKey
LED 1-64	5mm Red LED	C503B-RAN-CZ0C0AA2	0.19*64	DigiKey
Q0-7	Fast Switching MOS-FET	2N7000TACT-ND	0.74*8	DigiKey

Table 2: Component Choice Justification

Label in schematic	Functional reason for selecting this component
8_Ohm1	Main output of audio signal to be perceived by the listener.
C1	Used in part of low pass filter after the level shifted input
C3	Used in part of low pass filter as part of band pass filter before speaker output.
C4	Unused but included to change gain on LM386 if necessary during testing.
C6	Part of second order active low pass filter.
C7	DC blocking capacitor before the LM386 uses its internal voltage reference to shift the signal up from being purely AC.
C8	Part of second order low pass filter.
C9	DC blocking capacitor used to protect speaker from DC current burning it out.
P1	Echo Coefficient, never used as this was not implemented in software.
P2	Echo Time Offset, never used as this was not implemented in software.
P3	Pitch offset, used as an input to the STM32 board which would then change the pitch offset based on the position of the potentiometer.
P4	Coarse volume control. Used to make large changes to volume level using voltage splitting of the audio signal
P5	Fine volume control. Used to make small changes to volume level using voltage splitting of the audio signal.
R1	Current regulating resistor for pull up switch.
R2	Current regulating resistor for input of pull up switch to STM board.
R3	Current regulating resistor for input of pull up switch to STM board.
R4	Current regulating resistor for pull up switch.
R5	Current regulating resistor for input of pull up switch to STM board.
R6	Current regulating resistor for pull up switch.
R7	Current regulating resistor for pull up switch.
R8	Used with R9 for voltage adder between the reference voltage and the signal voltage.
R9	Used with R8 for voltage adder between the reference voltage and the signal voltage.
R10	Current limiting resistor for negative feedback on MCP602 to ground.
R11	Current limiting resistor on high side of LM385 voltage reference.
R12	Used to set the output voltage swing for the 100k volume control potentiometer. This is 10k Ohm so that the swing is 91% for the coarse volume control.
R13	Used to set the output voltage swing for the 100k volume control potentiometer. This is 1M Ohm so that the swing is 9% for the fine volume control.
R14	Impedance matching resistor for band pass filter before speaker output. The speaker output is about 8Ohm so a 10Ohm resistor was used to match it.
R15	Input resistor for second order active low pass filter.
R16	Resistor between non-inverting and negative feedback of second order active low pass filter.
R17	Current limiting resistor for passive low pass filter after level shifted output to the board. Used to reduce noise.
R18	Used for current limiting resistor to ground of negative feedback.

R19	Current limiting resistor for input to non-inverting side of LM386 audio amplifier.
R20	Current regulating resistor for input of pull up switch to STM board.
R21	Used for closed loop negative feedback on buffer or voltage follower used to isolate voltage reference from the AC signal voltage.
R22	Closed loop feedback for voltage adding voltage follower to insulate board from direct input.
R23	Current limiting resistor for potentiometer.
R24	Current limiting resistor for potentiometer.
R25	Current limiting resistor for potentiometer.
R26	Not used but was laid out on PCB so the LM386 gain could potentially be changed.
R27	Used in calibration of voltage reference.
R28	Used in calibration of voltage reference.
STM1	The main processing unit for the entire project used for all logical processing and for signal processing and transforming using an FFT and custom software described above.
SW1	Switch used for user input.
SW2	Switch used for user input.
SW3	Switch used for user input.
U1	This was used for both the level shifter voltage follower and for the reference voltage follower to insulate the reference voltage circuit from the AC signal. Then it was also used to insulate the board form the imputed signal.
U2	This was used for a second order low pass filter to get rid of quantitation error by filtering out the drastic steps in the voltage signal.
U3	This was the LM386 Audio operational amplifier used to cleanly amplify the audio signal to drive the speaker.
VR1	This is the LM385 used for a voltage reference that was then used for the level shifting of the input signal voltage.
LED 1-64	These were used for an 8 by 8 pixel display created on the LED board.
2N7000	These op amps were used to ground a specific row of the LED display so that a specific row could be displayed at a specific time.

V. Circuit Schematic & PCB

The following figures detail the schematic for the main board (see Figure 5) and the LED board (see Figure 6). The PCB layout for the main board (see Figure 7) and the LED board (see Figure 8) are detailed. The bare PCB for the main board (see Figure 9) and the LED board (see Figure 10) are shown. The populated PCB for the main board (see Figure 11) and the LED board (see Figure 12) are shown.

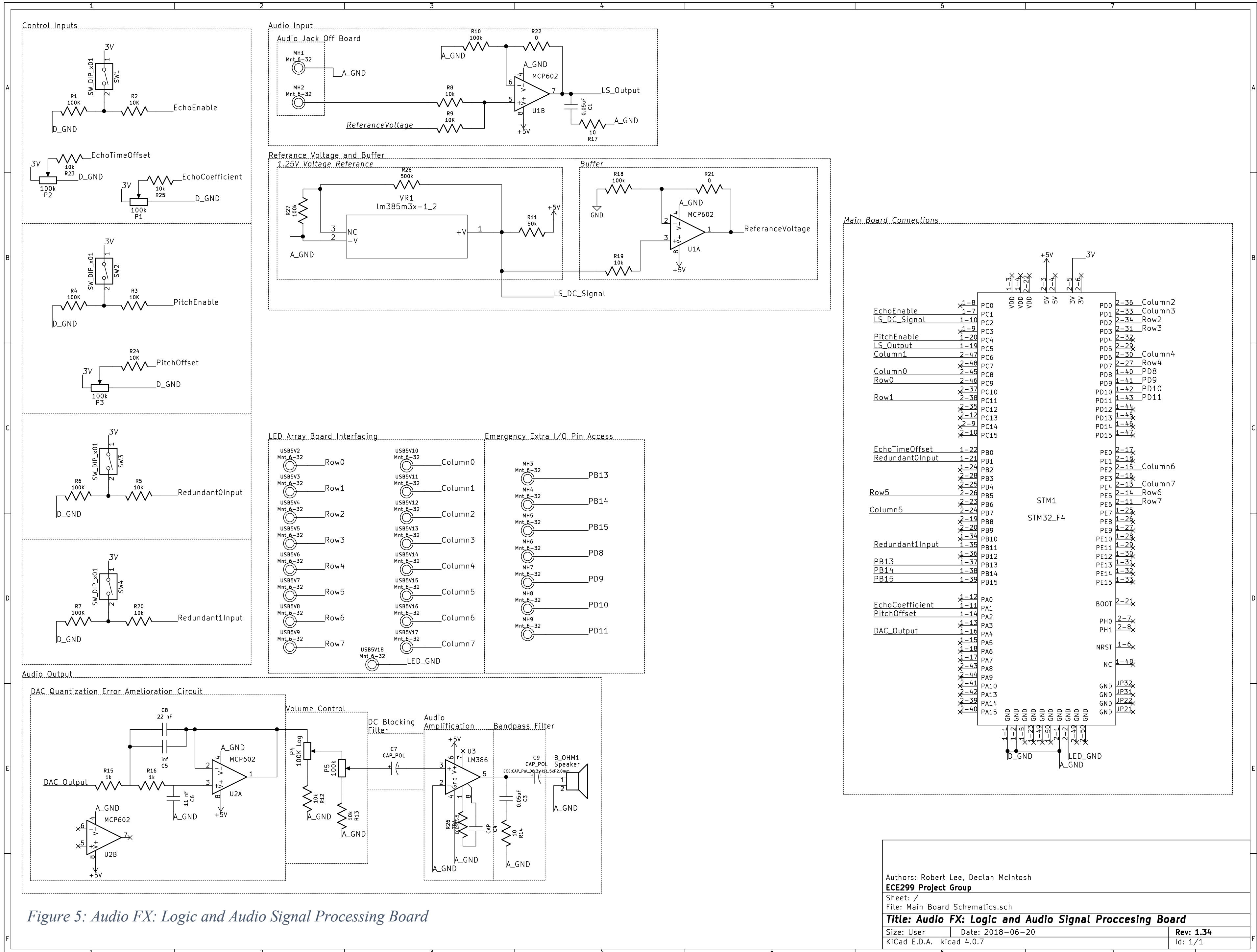

Figure 5: Audio FX: Logic and Audio Signal Processing Board

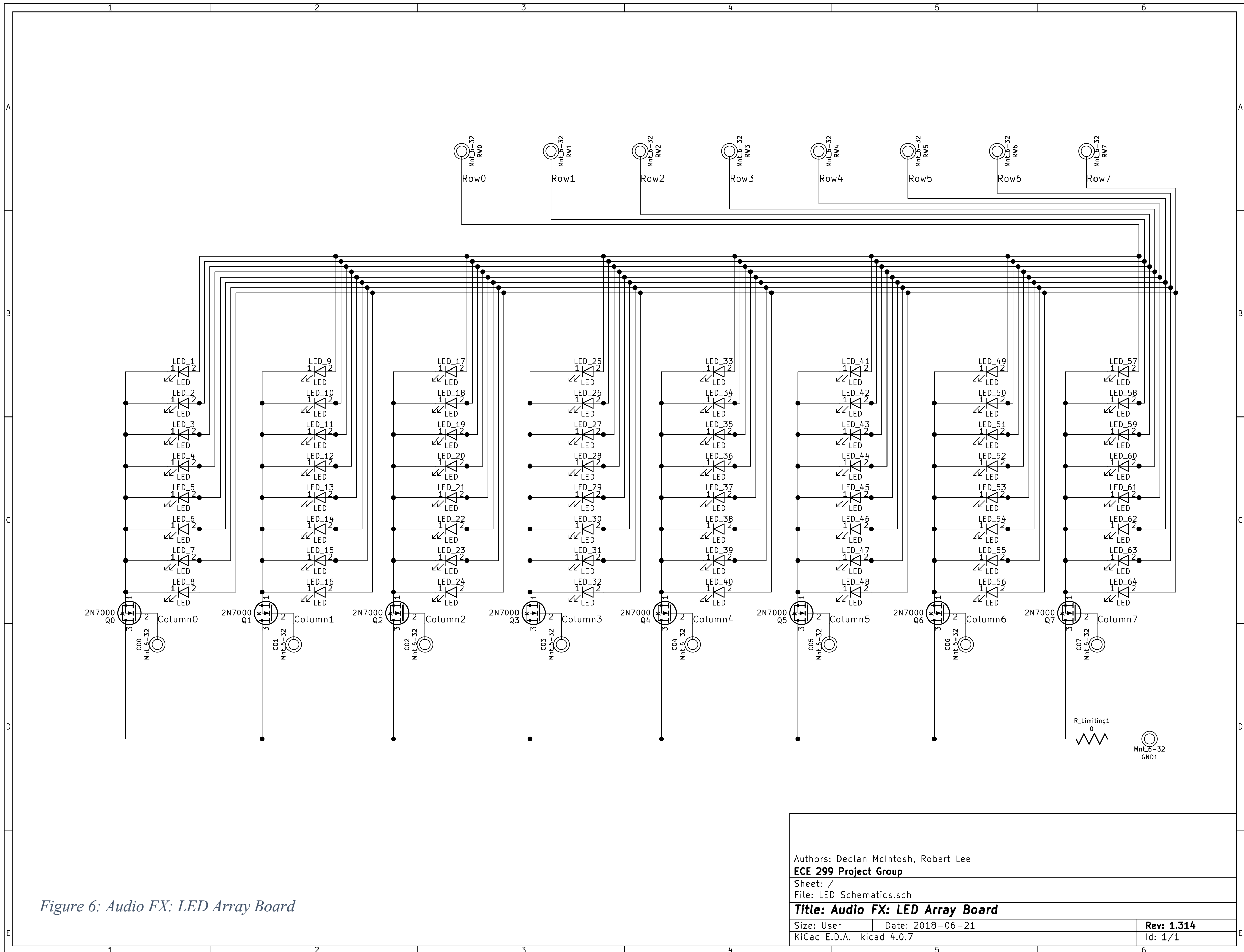

Figure 6: Audio FX: LED Array Board

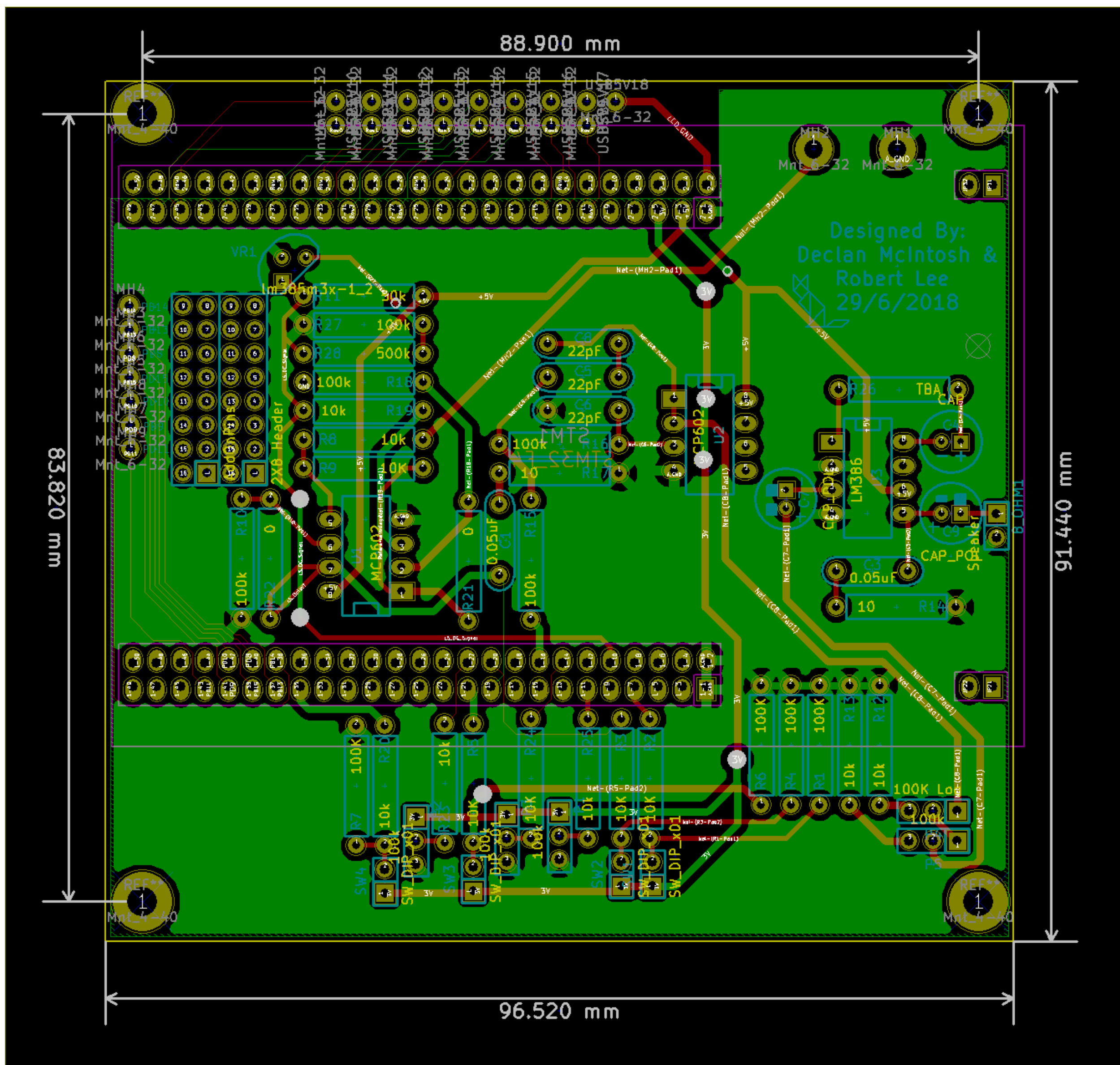

Figure 7: Logic and Audio Signal Processing Board PCB Layout

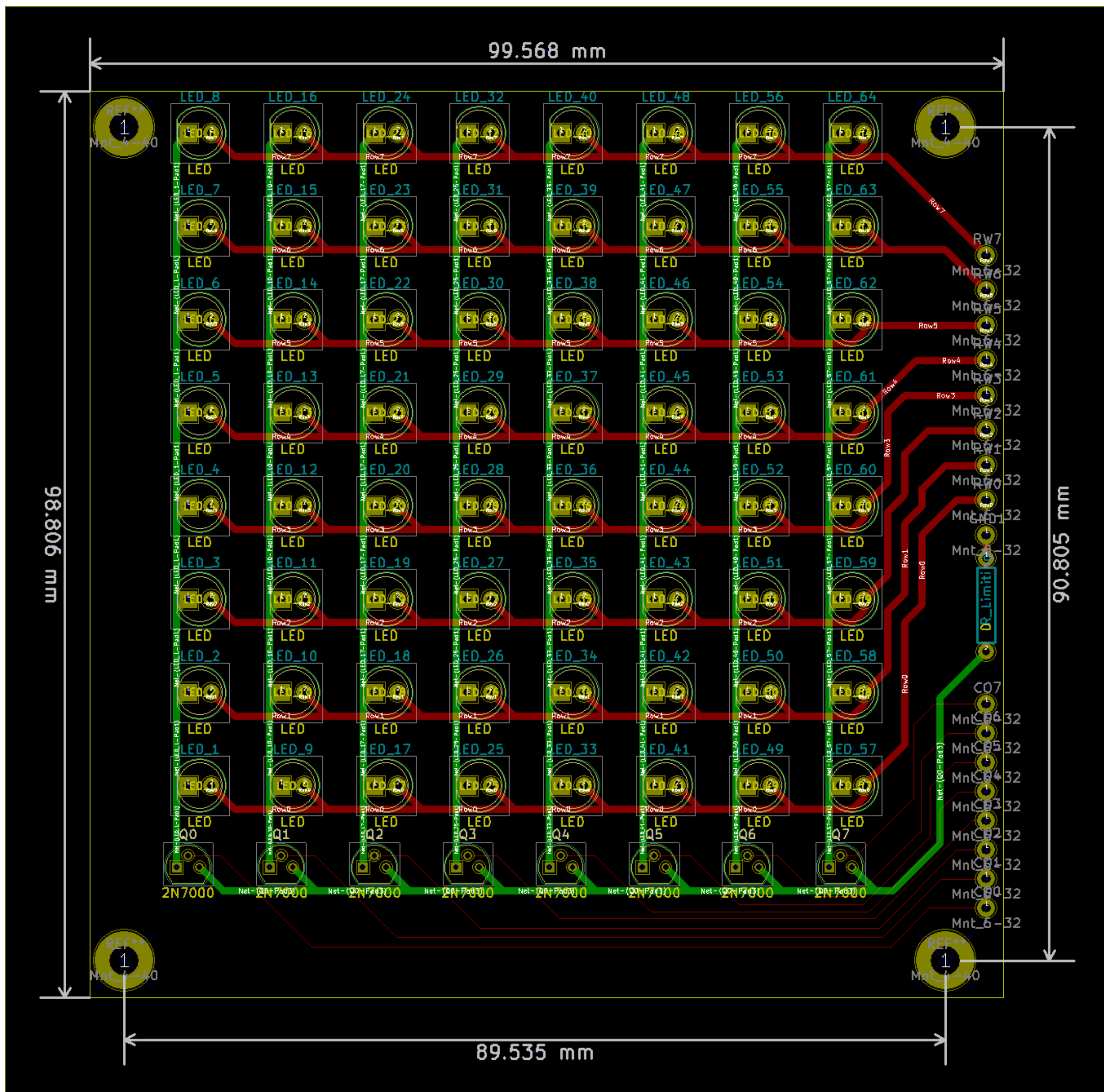

Figure 8: Audio FX: LED Array Board PCB Layout

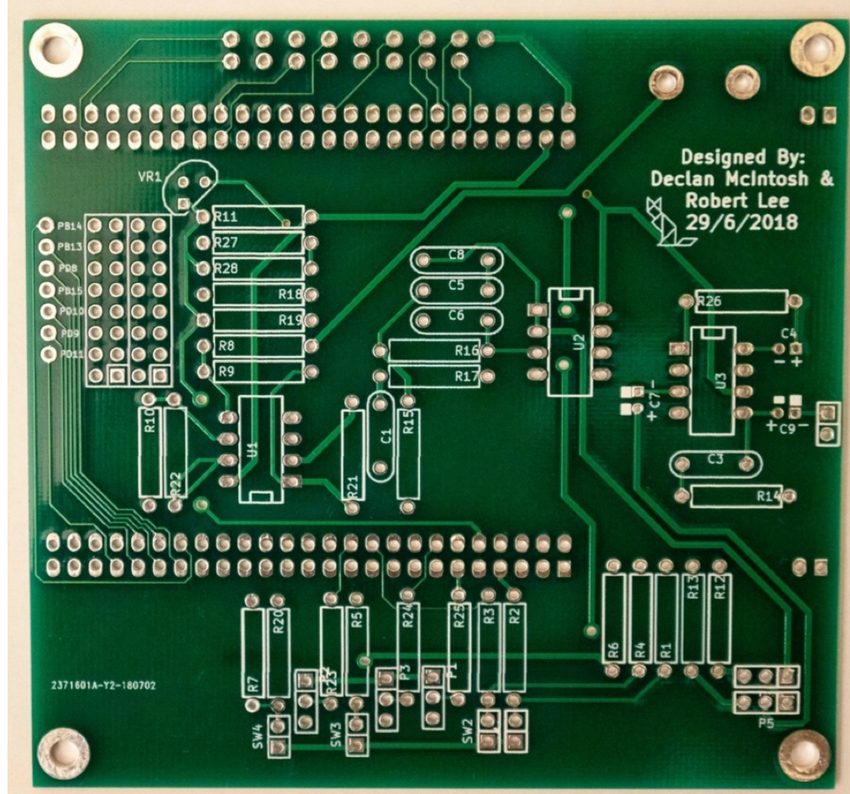

Figure 9: Logic and Audio Signal Processing Board Bare PCB

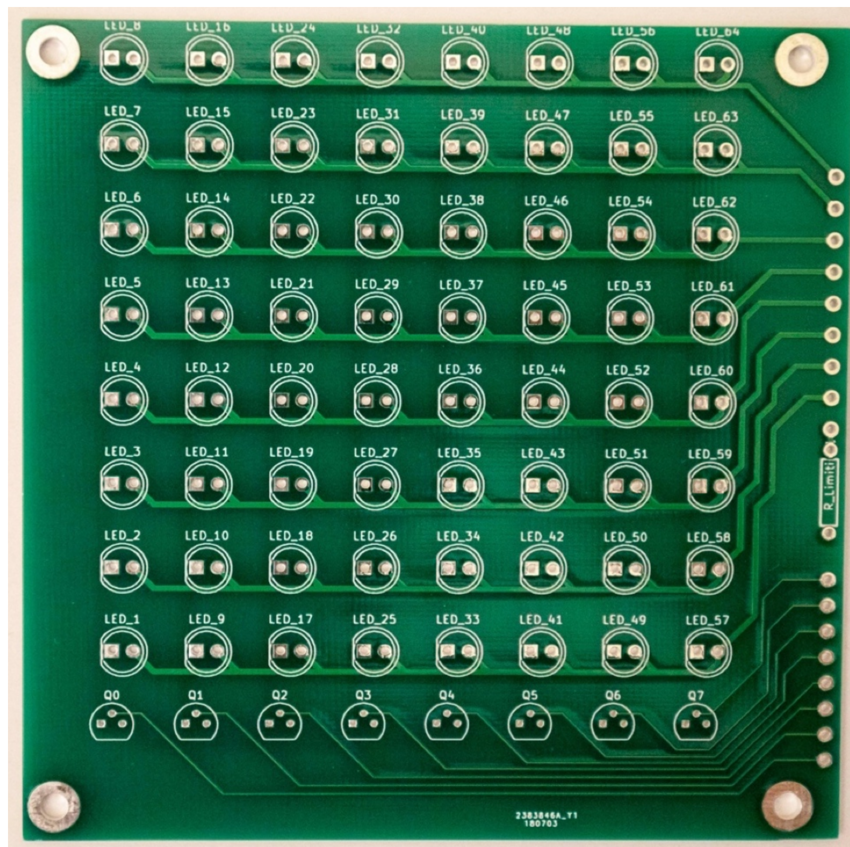

Figure 10: Audio FX: LED Array Board Bare PCB

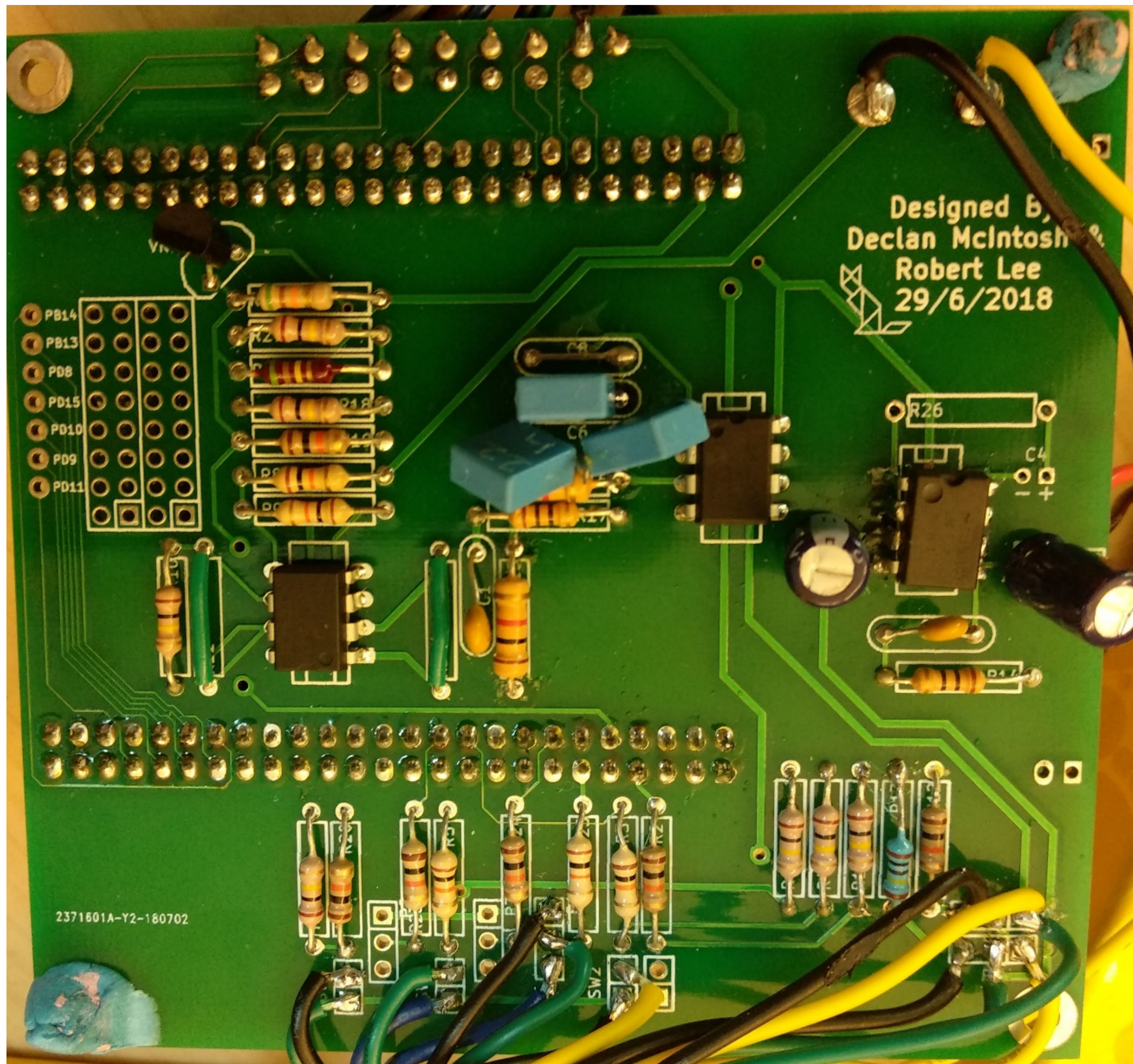

Figure 11: Logic and Audio Signal Processing Board Populated PCB

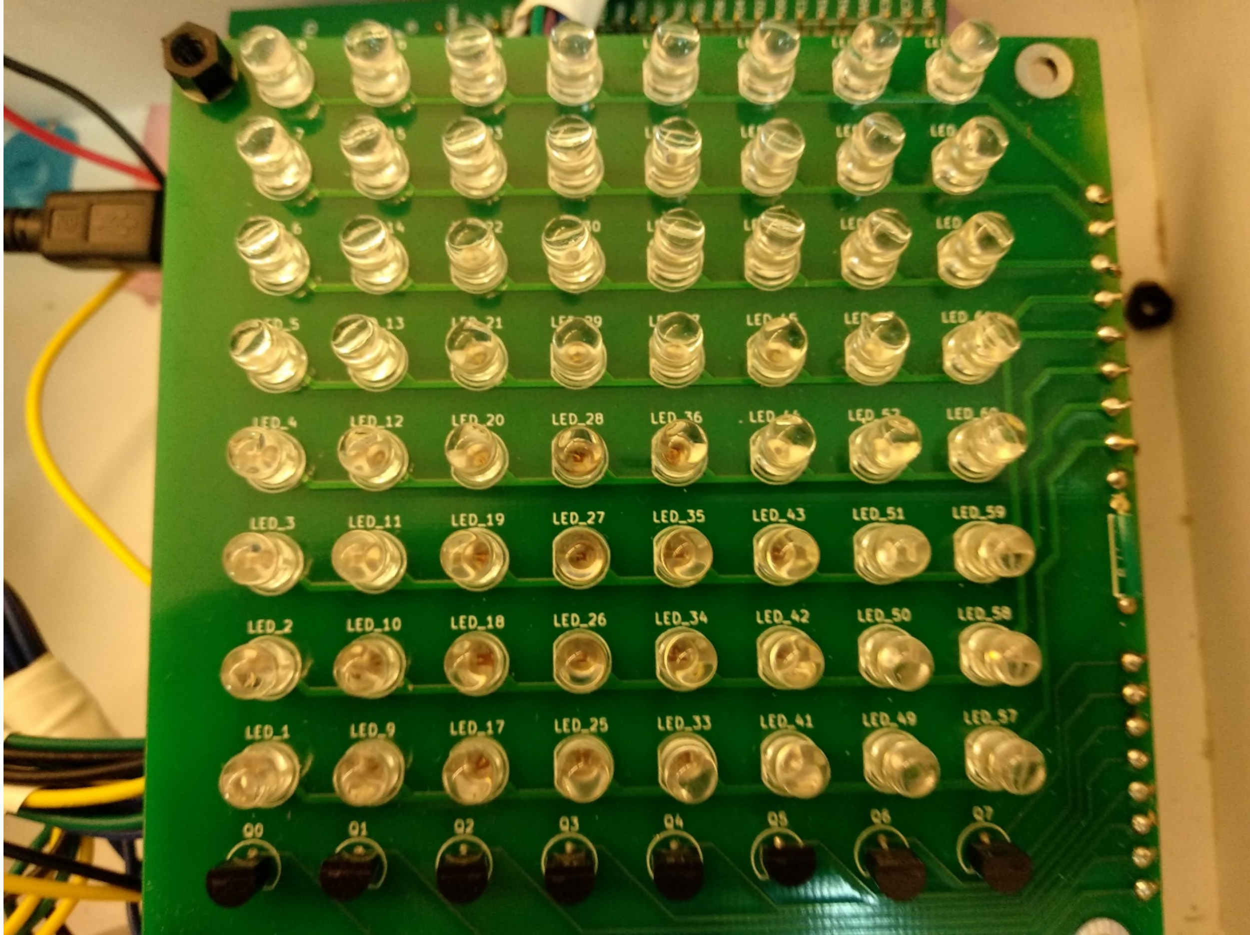

Figure 12: LED Array Board Populated PCB

VI. Testing & Validation

The testing and validation of the audio effects board required the use of the following equipment:

- Oscilloscope with two inputs and probes
- Function/signal generator
- DC power source
- Multimeter
- Audio source
- 3.5mm male connector
- Laptop

The testing and validation of the LED board required the use of the following equipment:

- DC power source
- Laptop

Function Test Plan of Audio FX 299 Project						
Test	**Test Purpose**	**Test Procedure**	**Test Pass Specifications**	**P/F**	**Tested By**	**Date**
Tone Input	This test is performed to verify the integrity of the frequency of inputs to the frequency of outputs.	Play a known sinusoidal frequency of 440Hz signal into the input audio jack and note the tone output relative to the tone output on a known functional system output. Using a normal audio input through an AUX cable.	There should be no audible difference between the tone played through the known working and DUT unit.	Pass	DM	7/14/2018
Bass Heavy Song	This test is performed to verify the integrity of the bass heavy song inputs have proper, minimally distorted outputs.	Play a song with a bass-heavy track in the DUT. Using a normal audio input through an AUX cable.	Verify the song sounds correct, no noise, distortion or other anomalies should be noted in playthrough of the song.	Pass	DM	7/14/2018
Treble Heavy Song	This test is performed to verify the integrity of the treble heavy song inputs have proper, minimally distorted outputs.	Play a song with a treble-heavy track in the DUT. Using a normal audio input through an AUX cable.	Verify the song sounds correct, no noise, distortion or other anomalies should be noted in playthrough of the song.	Pass	RL	7/14/2018
Solid Display, Current and Brightness	This test is used to verify the current draw of the display in the worst case is such that it does not exceed expected limitations	Run the `Display_All_On` function provided in the source code, note display condition during test.	The display should be entirely lit, the entire display should have the same brightness, the LEDs should be completely visible and clear. Note any current limiting by the STM board or USB supply.	Pass	RL	7/14/2018
Solid Display, Display Temperature	This test is to verify the thermal dispersion of the display unit.	Run the `Display_All_On` function provided in the source code, note display condition during test. Note temperature qualitatively.	The display should be entirely lit, the temperature of each led should be consistent and minimal, this condition must be shared with all traces related to the display on both the LED and Audio FX boards, as well as the wires connecting the two boards.	Pass	DM	7/14/2018

Single Pixel Addressability	This test is to verify accurate addressing and scanning of the display.	Run the `Display_Scan_Across_LEDs` and press the button on the board to start the test, then note the direction of the cycling, where the cycling started and where the cycling finishes.	The cycling should start in the top left (when the button is pressed) of the display as viewed from the I/O ports then cycle across each row from top row to bottom row. The final display state is all pixels are off.	Pass	RL	7/14/2018
Wake Up Initialization	To test the time between the initialization starts and the board is completely initialized.	Run code with a breakpoint at the beginning of code. Then resume the running of code while starting a timer, then note the time before a signaling LED on the board lights up signifying initialization completion.	This time should be less than 100ms.	Pass	DM	7/14/2018
Volume Controls	This test verifies the functionality of the coarse and fine volume controls.	Run music though the system, where both controls are at their maximum positions. Turn the fine control down and note the change in volume. Turn the coarse knob down and note the change in volume. Then turn coarse knob until a switch is felt. Note if the music is still playing.	The fine control should have a relative swing approximately of 10% of maximum volume. Coarse control should swing approximately 90% of the relative beginning and ending volume when moved its entire range. When the coarse knob is switched all the way to its lowest position the music should stop playing.	Pass	RL	7/14/2018
I/O Button Testing	This test verifies the functionality of the button inputs.	Run `Test_Button_State` in the provided source code. Press buttons in random orders several times and note changes to the LED display.	Any button pressed in any order should toggle the LED board all on or off. The buttons should be responsive and never false trigger due to debouncing issues.	Pass	DM	7/14/2018
Level Shifter Testing	Verify the expected Level shifter functionality	Check the outputted voltage from the level shifter output pin with a voltmeter	The voltage should read approximately 1.24 V	Pass	RL	7/14/2018
2nd Order Low Pass Filter Testing	Verify the frequency response and cut off frequency is as expected	Input known sine wave at known frequencies sweep through the higher frequencies until the output wave is 70.7% of the input waves, measured on an oscilloscope	The approximate frequency at which the output wave is 70.7% of the input amplitude should be about 10.4 kHz.	Pass	DM	7/14/2018

Amplifier Testing	Verify the functionality of the output audio signal amplifier.	Input a known amplitude signal which will not cause clipping on the amplifiers output then observe the output signal amplitude.	The amplitude of the output should be approximately 20 times larger than the amplitude of the input signal.	Pass OK	RL	7/14/ 2018
State Machine Testing	Pressing the state buttons should transition between the audio effect states well.	Press the pitch shift button and note the output of the speaker. Then press the pitch shift button and then the echo button observing output. Then press the pitch shifting button again observing output.	The first observation should be the pitch shifted audio, then the next observation should yield an echo to the output. Finally the third observation should not pitch shift and just echo.	Pass	DM	7/14/ 2018
Display state button.	Pressing the display state button should accurately display the state.	Press the display button in each of the four possible states, echo pitch, echo and pitch, and no state.	The display should show running text of the state for each of these states accurately when pressed. If there is no state, space invaders should slide across the screen.	Pass	RL	7/14/ 2018
Pitch shift up and down.	Testing that the pitch shift fully works in both up and down pitch shifts.	While playing music press pitch shift on, then turn the pitch shift knob, noting the pitch shifting in the song up and down.	The pitch should shift up when the knob is turned to the right and down when turned to the left. Further no pitch shifting should occur when it is centered.	Pass	DM	7/14/ 2018
Echo	Test that the echo occurs when the state is enabled.	Enable the echo state while playing music then suddenly pause the input music and note the output heard after the music is disconnected	The final 1 second of music played before the pause should be echoed several times showing that the echo is functioning during normal operation.	Pass	RL	7/14/ 2018
Play music without effect	Note the functionality of playing music though the board.	Connect an audio input playing music to the 3.5mm input jack and note the output.	The output should be the expected song undistorted and with minimal noise.	Pass OK	DM	7/14/ 2018
Volume changing	Verify the volume can be changed in real time.	Connect an audio input playing music to the 3.5mm input jack; note the output during movement of the volume knob.	The volume should change significantly as the knob is turned.	Pass	RL	7/14/ 2018
Display testing	Verify functionality of the display.	Press the mode display button when no effect is being affected note the display after this button is pressed over several pressings.	The display should show space invaders going across the display. No LED should be on afterwards and the images should be clear.	Pass	DM	7/14/ 2018

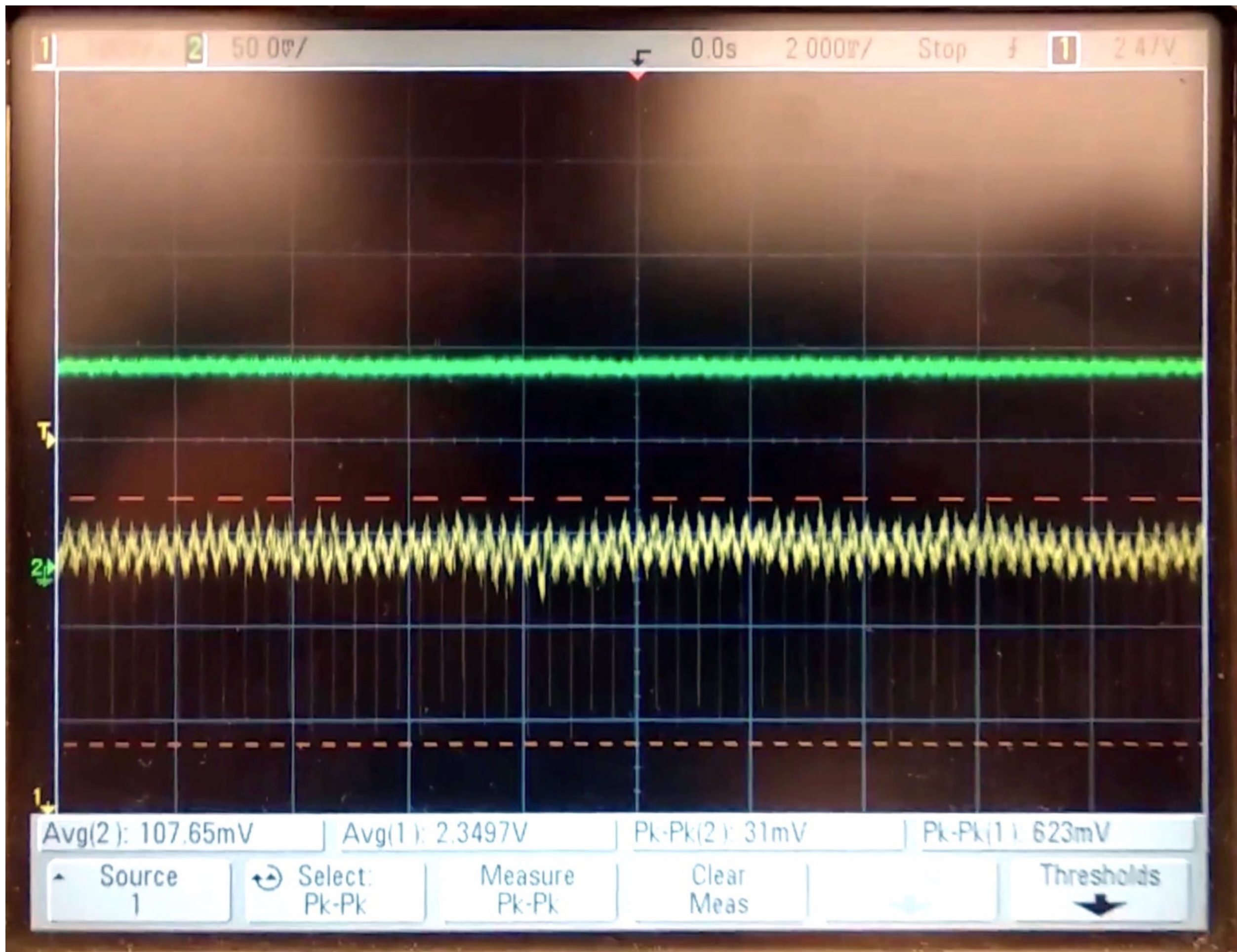

Figure 13: Oscilloscope display of the audio amplifier scaling the input signal by approximately 20 times. Yellow corresponds to output and green corresponds to input.

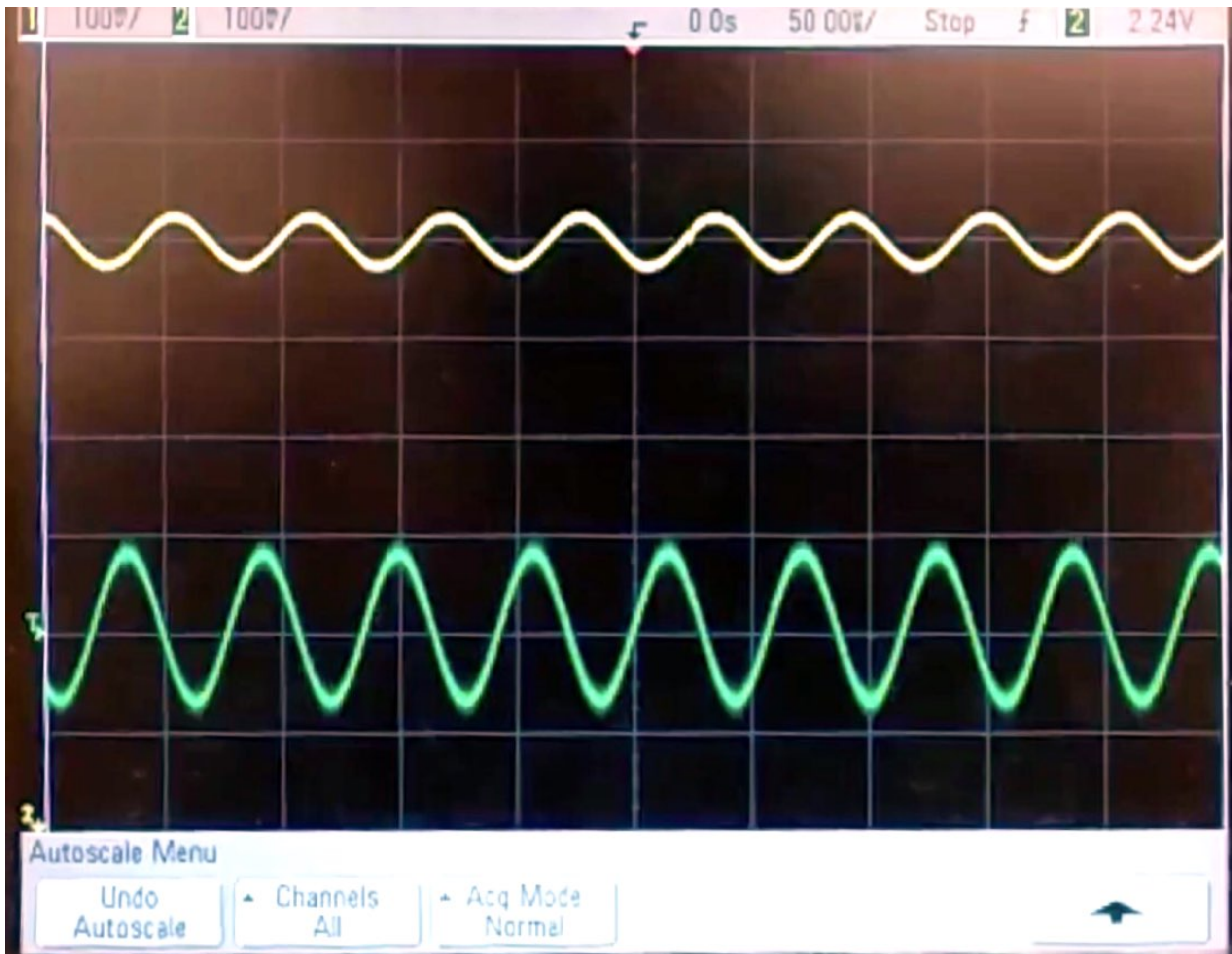

Figure 14: Oscilloscope display showing the attenuation of a signal past the cut-off frequency of 10.4kHz. Green corresponds to input and yellow corresponds to output on the same scale.

VII. Conclusion & Recommendations

All major objectives of functionality were achieved for the final project, with some parts of systems having minor issues and others exceeding requirements adding functionality. The project's analog audio subsystems including the second order low pass filter to remove quantization error, the level shifting input, and the audio signal amplifier using a LM386 functioned as expected with some marginal deviation from theoretical values. Some noise was noted in the final build of the FX project. This noise is expected to be caused by noise in the power supply. Further the amplification of the audio signal was approximately 15 times gain which was lower than the expected 20 gain of the configuration as designed, giving a quiet sound to the output. The display requirements were met entirely with an 8 by 8 LED matrix for display. Further the requirements were met with two linear potentiometers used as voltage dividers of the input voltage signal which were used for coarse and fine volume control. However, as the potentiometers did not have plastic caps affixed, when a person would touch the potentiometers some noise would be introduced from the person. All software requirements were met. Button debouncing was performed using an ISR routine on a 5ms timer. Pitch shifting was implemented using potentiometer-controlled analog input to

determine the magnitude and direction of the frequency shift. A circular buffer was used to store and retrieve raw analog ADC values to implement echo. A finite state machine architecture was implemented to control which effects were being used at any given time. Finally, another circular buffer was implemented to display things on the LED array. Overall all expected and requested functionality was at least minimally met.

There were some issues with a lack of simulation software embedded into the KiCAD software. The wire used for the connection between the LED display and the main Audio FX board was 22AWG multi-strand wire and would often break when handled. This was an issue that would had taken a considerable amount of time to fix entirely so the breaks were fixed as they happened. During the implementation of pitch shifting there were some issues shifting up, as low frequencies resulting from the Fast Fourier Transform of the level shift offset, which would produce an impulse-like value in the low frequency range. As frequencies were shifted up, this impulse-like value would be shifted past the minimum frequency to be filtered out by the DC-blocking capacitor, causing it to be audible as a harmonic. This was solved by starting the shift at a higher frequency, so the impulse value would be unmoved.

There were several limitations to the final design of our Audio FX board. Firstly, the display buffer could only handle 10 seconds of frames at 25 FPS. This took up 2 KB of memory, which can be an issue given the relatively limited 192 KB total memory on the board being shared across an echo buffer, FFT dependencies, and FFT output bins [7]. The I/O, while functional for the required testing with only 3 pushbuttons, was relatively limited in potential for further functionality without a convoluted user interface. The major limitation of the final implementation of the project was the poor maximum volume only amplifying the outputted signal by 15 times gain which made the sound quiet.

If a successor to our first design was to be built some major changes would be made. First, another second-order active-low-pass filter would be placed on the input to the ADC of the STM board so that higher frequency signals on the input would not be sampled and cause aliasing issues as it wraps around into the maximum frequency of 8 kHz signals the STM board is sampling at. This would help improve general sound quality of the audio signal. To further improve the audio signal a capacitor should be placed between power and ground to reduce ripples in the voltage. Alternatively, all the amplifiers could be powered off a separate voltage regulator which would help provide a clean power source. This is expected to reduce the notable noise assumed to be caused by a noisy voltage source. Knowing that the board is not being put in a manufactured case, the components would be chosen as surface mounts rather than mounting holes for wires to connect to offboard components. Some changes should also be made to confirm a better gain on the audio amplifier to make the music experience much better to the ear. These changes would improve the sound quality and make the music listening experience much better. Additionally, using plastic caps on the potentiometers would prevent capacitance noise on the audio line. This noise is caused when the conductive metal exterior of the potentiometer is touched by a capacitive disturbance like a person who is not properly grounded. Finally, the number of pull-up resistors could be reduced by using the board’s internal pull-up or pull-down resistors, reducing the cost and complexity of the PCB.

References

References cited should be easily obtainable. Please refrain from using sources that are not trust worthy. Preferable to refer text books and works published by international organizations of repute. It is mandatory to cite the source from which you learnt about a concept/idea. It is not acceptable to copy and paste images. They have to be re-drawn. References must be in IEEE standard format (https://ieee-dataport.org/sites/default/files/analysis/27/IEEE%20Citation%20Guidelines.pdf).

[1] "USB Power Delivery," Universal Serial Bus. [Online]. Available: http://www.usb.org/developers/powerdelivery/. [Accessed: 29-Jul-2018].

[2] Dr. T. Ilamparithi. ECE 299. Class Lecture, Topic: "STM32F4 Discovery Board." Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, June 13, 2018.

[3] Texas Instruments, "LM386 Low Voltage Audio Power Amplifier," SNAS545C datasheet, May 2004 [Revised May 2017].

[4] Texas Instruments, "LM185/LM285/LM385 Adjustable Micropower Voltage References," SNVS741F datasheet, Feb. 2000 [Revised April 2013].

[5] "Sallen-Key Active Butterworth Low Pass Filter Calculator," *Modulus Of Rupture Calculator - Load, Distance, Breadth, Depth*. [Online]. Available: http://www.calculatoredge.com/electronics/sk low pass.htm. [Accessed: 17-Aug-2018].

[6] "Digikey Electronics," *Digikey Electronics*. [Online]. Available: https://www.digikey.ca/. [Accessed: 17-Aug-2018].

[7] STMicroelectronics, *RM0090 Reference manual for STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM®-based 32-bit MCUs*, STMicroelectronics, April 2017. [Page 68]

[8] Robert Lee and Declan McIntosh, *AudioEffectsProject Repository*. [Github]. Available: https://github.com/robertklee/AudioEffectsProject

Appendix A – Enclosure

The design of the enclosure was intended to sit well on a desk and show off the vibrant display and the intuitive user interface (see Figure 15, Figure 16, and Figure 17). The speaker faces directly up to be omnidirectional in the sound stage produced by the speaker. The enclosure is expected to be made from plastic using M3 screws to mount both the LED and the main Audio FX boards to the enclosure. M6 screws are used to hold the bottom plate to the bottom of the enclosure. The top hole for the speaker is designed so some epoxy or other adhesive can be used to secure the speaker as the speaker used in the project does not have mounting screw holes. There is some translucent material which would be used in front of the LED board from the outside over the opening. This would likely be a clear plastic or plexi-glass substance. Then the input buttons and potentiometers are fit into the enclosure using M7 sizing fit. Once fit into the appropriate labeled holes in the interface paneling, they are held in place by the mounting hardware provided with the buttons and potentiometers. Finally, the audio input is placed at the side of the case where the 3.5mm jack will be placed internally and poke through the hole.

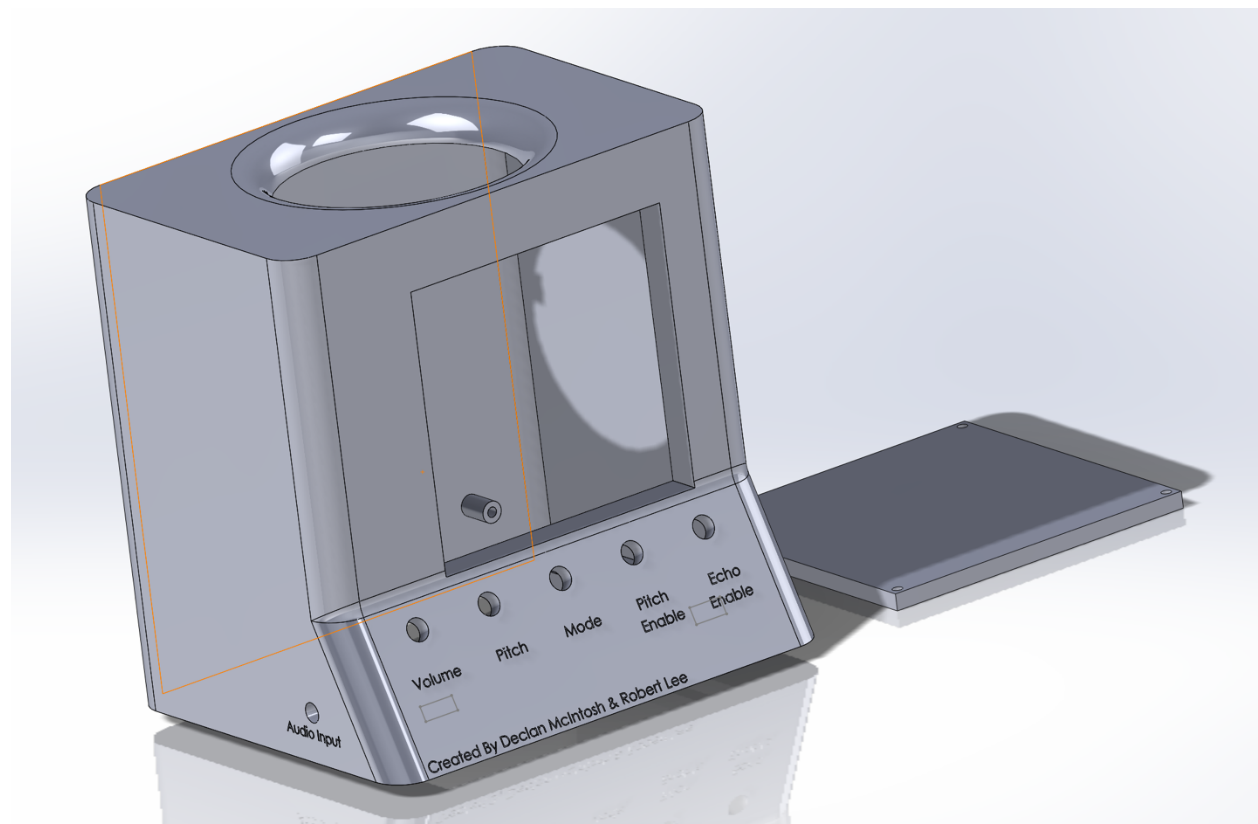

Figure 15: Enclosure for the Audio Effects board and LED display. Note the agronomical button placement on the face of the product.

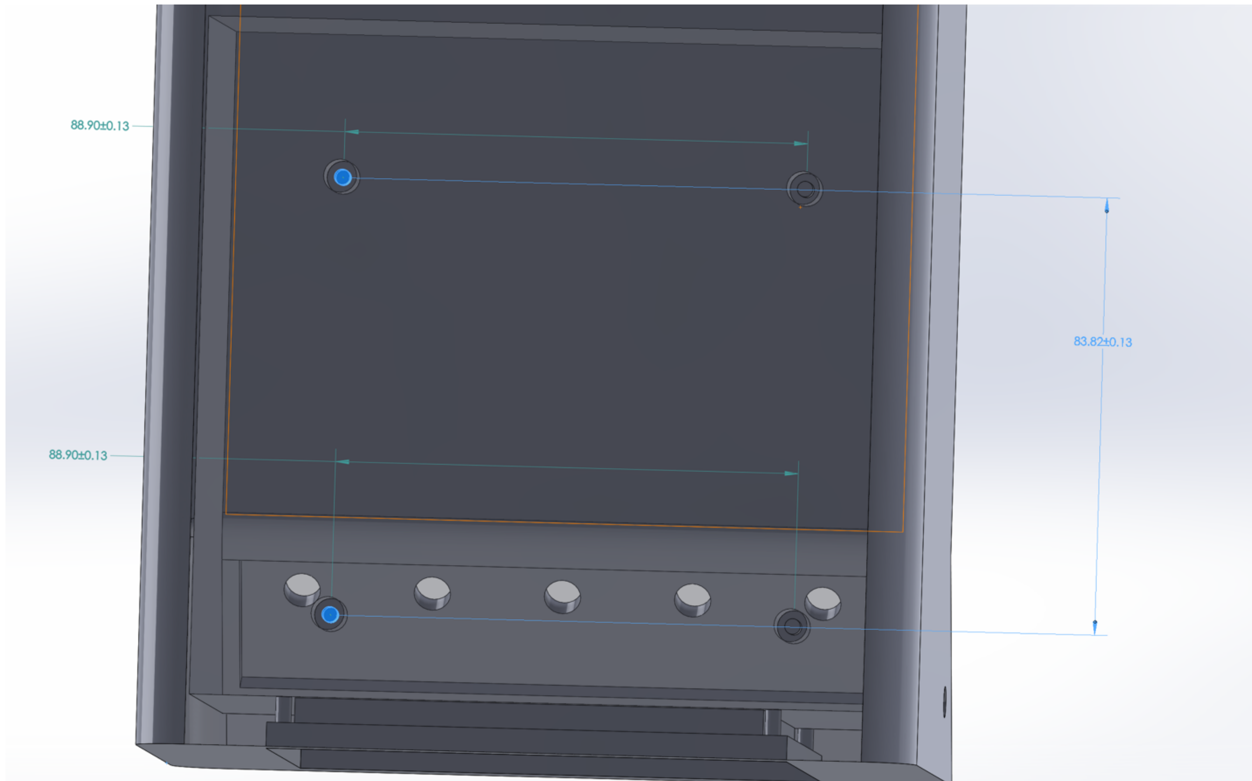

Figure 16: The rear of the enclosure, showing the mounting hole locations for the main circuit PCB.

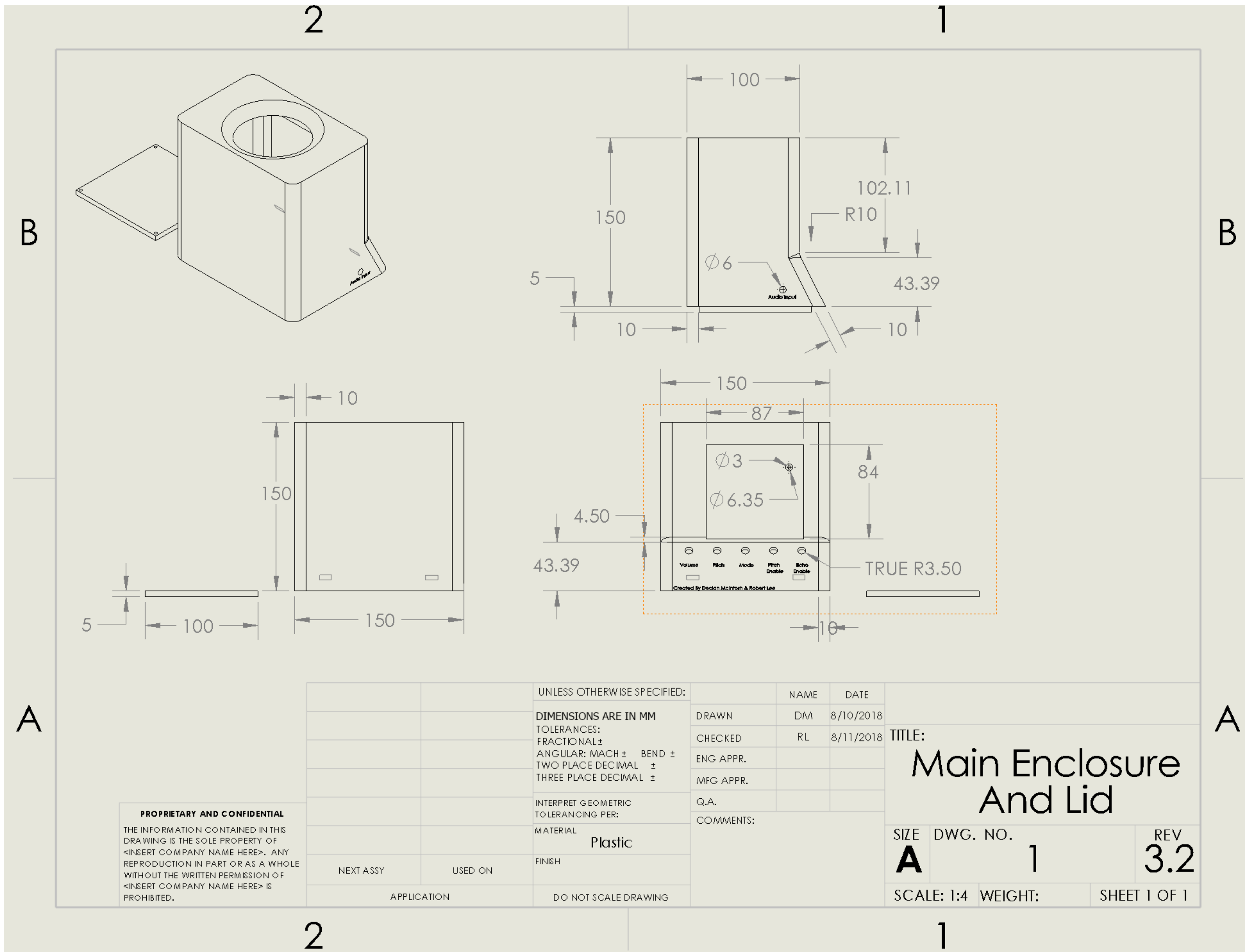

Figure 17: Multiple views of the enclosure designed using SolidWorks a

This page is intentionally left blank.

Appendix B – Gantt Chart for Hypothetical Projects

In this appendix, the planning for a theoretical project is described. This goal of this project would be to design a robot which can navigate a right-angled maze. This project is mechatronic in capacity so all disciplines, mechanical, firmware, and electrical design have been factored into the planning of this project. Two hypothetical scenarios have been considered around this project: A) The familiar STM32F4 Discovery board is used and B) a new, unknown TMS320F28035 Piccolo development board is used for the implementation. The Gantt charts are shown below (see Figure 18 for the STM32F4 Discovery board Gantt chart, and see Figure 19 for the TMS320F28035 Piccolo development board Gantt chart) to demonstrate the developed plans and their respective differences caused by the changed task parameters. The details of each task will be overviewed and not described in extreme detail as it is not pertinent to the differences introduced by the change in discovery boards and are subject to the specific details of the project.

Implementation using the STM32F4 Discovery Board

Firstly, 6 hours has been allocated for the high-level conceptual design to develop a general strategy for this task. Then 10 hours were dedicated to design each of the subsystems. The mechanical systems should be fairly easy due to the simple mechanical nature of the project, unless a unique robot design was chosen. However, due to the inexperience and lack of technical expertise our team exhibits in mechanical design equal time weighting has been given the mechanical design task. Near the end of the designing of the subsystems, time has been allocated for sourcing of all the required parts for the Prototype-0 build. After the required parts have been retrieved 5 hours has been allocated for populating the controlling PCB board as our team has a good amount of experience soldering. The longest expected task is to implement the planned software into the STM board with 12 hours as the software for maze solving and control of the mechanical systems is expected to be quite difficult to write and debug. Next, time has been allocated for testing of the P0 design to highlight any flaws which must be solved before the final project submission. Finally, time, which may not be required by the project outline but is beneficial, has been allocated for a second P1 redesign and rebuild to address any issues found during testing.

Implementation using the TMS320F28035 Piccolo development board

Some significant changes were made to the allocated time during the design phase of the plan. Time has been allocated for learning the new development board with regards to its electrical and software architecture. This is required so that the electrical and software design efforts can be completed more smoothly and design around the qualities of the Piccolo board. This also delays the start of the electrical and software design process. Furthermore, time has been allocated for implementing the software for the Piccolo board. Small amounts of time have also been added to the redesigning of the P0 to the P1 version as there are more expected errors when working with an unfamiliar development board. Overall the change of board has added 27 hours of total labour to the project.

This page has intentionally been left blank.

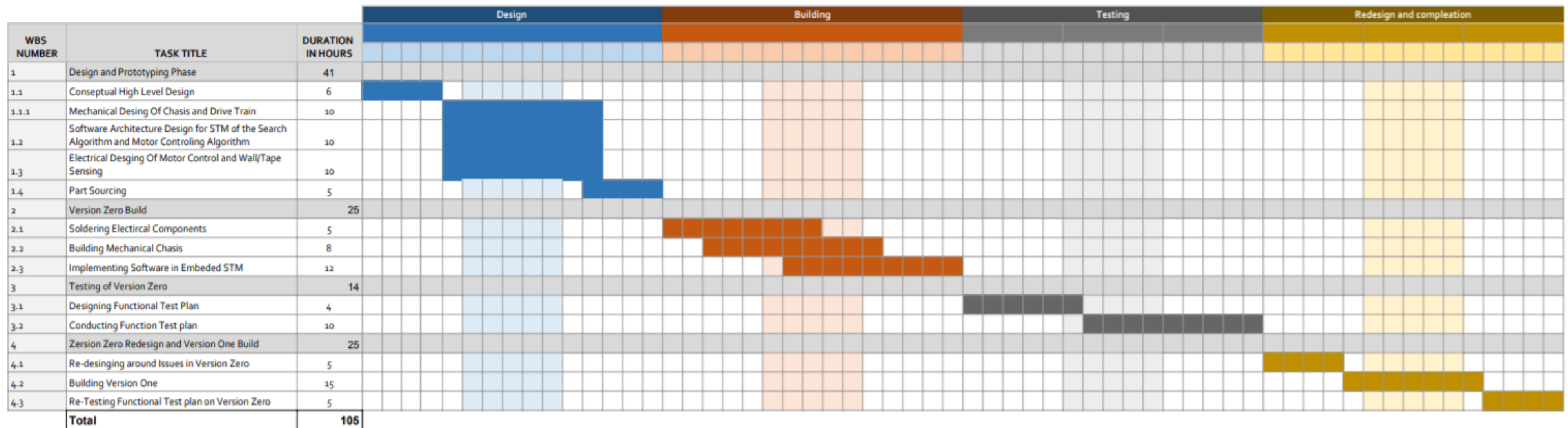

Figure 18: Gantt chart for first hypothetical scenario: design using STM32F4 Discovery Board

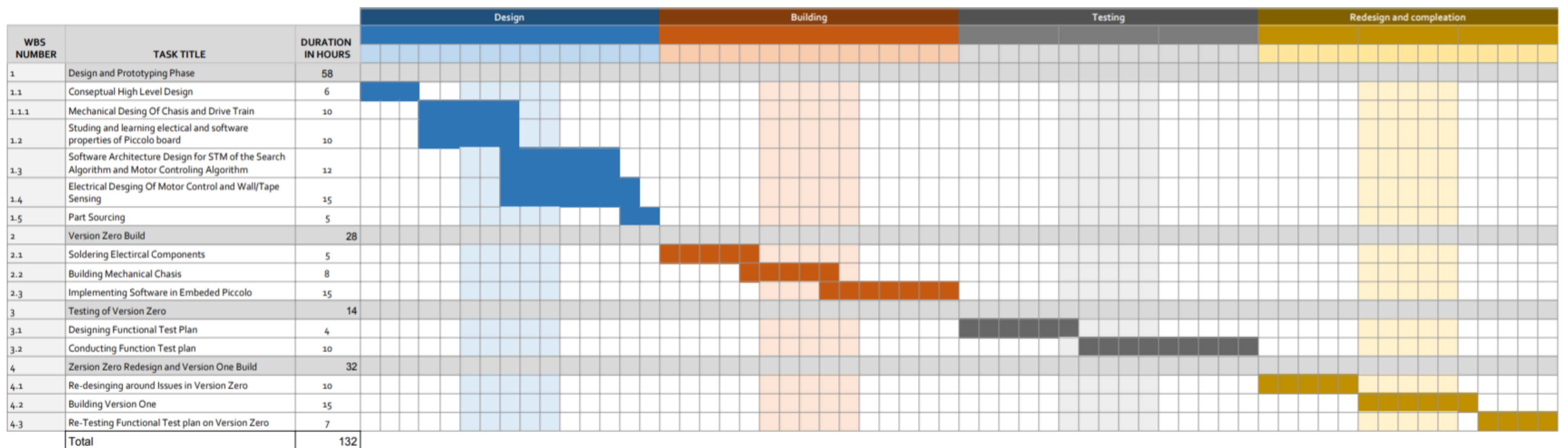

Figure 19: Gantt chart for second hypothetical scenario: design using TMS320F28035 Piccolo development board

This page has intentionally been left blank.

Appendix C – Sample Code

The full source code is available on Github [8].

```
// Copyright (c) 2018 Robert Lee, Declan McIntosh
// University of Victoria ECE 299 Design Project

/*
 * This file is part of the �OS++ distribution.
 *   (https://github.com/micro-os-plus)
 * Copyright (c) 2014 Liviu Ionescu.
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without
 * restriction, including without limitation the rights to use,
 * copy, modify, merge, publish, distribute, sublicense, and/or
 * sell copies of the Software, and to permit persons to whom
 * the Software is furnished to do so, subject to the following
 * conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 */

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

#include <stdio.h>
#include <stdlib.h>
#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"
#include "stm32f4xx.h"
#include "ctype.h"
#include <sys/stat.h>
#include "stm32f4xx_hal.h"
#include <string.h> //for memcpy

#include <hamming.h>
#include "stm32f4xx_hal.h"
#include "math.h"

#include "arm_math.h"
#include "arm_const_structs.h"
#include "main.h"
#include "hamming.h"
#include "windowing_fft.h"
#include "AudioChip.h"

// ----------------------------------------------------------------------------
//
// Standalone STM32F4 empty sample (trace via DEBUG).
//
// Trace support is enabled by adding the TRACE macro definition.
// By default the trace messages are forwarded to the DEBUG output,
// but can be rerouted to any device or completely suppressed, by
// changing the definitions required in system/src/diag/trace_impl.c
// (currently OS_USE_TRACE_ITM, OS_USE_TRACE_SEMIHOSTING_DEBUG/_STDOUT).
//

// ----- main() ---------------------------------------------------------------

// Sample pragmas to cope with warnings. Please note the related line at
// the end of this function, used to pop the compiler diagnostics status.
```

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wmissing-declarations"
#pragma GCC diagnostic ignored "-Wreturn-type"

/**
 * LED MATRIX LAYOUT:
 * ~~~~~~~~~~~TOP OF LED MATRIX~~~~~~~~~~
 * Columns: 0   1   2   3   4   5   6   7
 * Rows:
 *      0
 *      1
 *      2
 *      3
 *      4
 *      5
 *      6
 *      7
 */
#define NUMBER_OF_LEDS          (64)
#define REFRESH_RATE            (5000)//5000 will reduce the high pitched noise // this will be multiplied
by 64 since there are 64 LEDs
#define FRAMES_PER_SECOND       (25) // this number should be a factor of REFRESH_RATE
#define BUFFER_SIZE_SECONDS     (10) // number of seconds for which the buffer stores data
// NOTE: The following row/col are NOT on the same bus
#define ROW_0                   (GPIO_PIN_4)
#define ROW_1                   (GPIO_PIN_2)
#define ROW_2                   (GPIO_PIN_7)
#define ROW_3                   (GPIO_PIN_6)
#define ROW_4                   (GPIO_PIN_1)
#define ROW_5                   (GPIO_PIN_0)
#define ROW_6                   (GPIO_PIN_6)
#define ROW_7                   (GPIO_PIN_8)
#define COL_0                   (GPIO_PIN_9)
#define COL_1                   (GPIO_PIN_11)
#define COL_2                   (GPIO_PIN_2)
#define COL_3                   (GPIO_PIN_3)
#define COL_4                   (GPIO_PIN_7)
#define COL_5                   (GPIO_PIN_5)
#define COL_6                   (GPIO_PIN_5)
#define COL_7                   (GPIO_PIN_6)
#define BUTTON_1                (GPIO_PIN_11)
#define BUTTON_2                (GPIO_PIN_4)
#define BUTTON_3                (GPIO_PIN_1)

#define LEFT_TO_RIGHT           (0)
#define RIGHT_TO_LEFT           (1)
#define TOP_TO_BOTTOM           (2)
#define BOTTOM_TO_TOP           (3)

/**
 * Timer usage documentation:
 * TIM2 - Generating frequency bars to display on LED array
 * TIM3 - polling of all button inputs, and debouncing
 * TIM4 - LED board drawing
 * TIM5 - FFT on input signal
 */
#define TIM2_PRIORITY           (7)
#define TIM3_PRIORITY           (10)
#define TIM4_PRIORITY           (5)
#define TIM5_PRIORITY           (0)
#define NUM_OF_COLS             (8)

#define NO_EFFECT               (0)
#define ENABLE_ECHO             (1)
#define ENABLE_PITCH_SHIFT      (1)

#define ECHO_BUFFER_SIZE        (16384)
#define ECHO_DAMPING            (0.35)

volatile char previous_button_reading_PA0 = 0;
volatile char button_state_PA0 = 0;
volatile char previous_button_reading_PB11 = 0;
volatile char button_state_PB11 = 0;
volatile char previous_button_reading_PC4 = 0;
```

```
volatile char button_state_PC4 = 0;
volatile char previous_button_reading_PB1 = 0;
volatile char button_state_PB1 = 0;

char previous_state_PB11 = 0;
char previous_state_PC4 = 0;
char previous_state_PB1 = 0;

volatile char current_frame[NUM_OF_COLS];
volatile char display_buffer[FRAMES_PER_SECOND * BUFFER_SIZE_SECONDS][NUM_OF_COLS];
const int buffer_length = FRAMES_PER_SECOND * BUFFER_SIZE_SECONDS;
volatile int buffer_head = 0; // points to front of buffer
volatile int buffer_tail = -1; // points to next available spot

volatile char current_row = 0;
volatile char current_col = 0;
volatile int current_frame_number = 0;
const int times_to_repeat_frame = REFRESH_RATE / FRAMES_PER_SECOND;

int LED_Array_State = 0;
int pitch_shift_state = NO_EFFECT;
int echo_state = NO_EFFECT;

volatile int pitch_shift_offset = 0;

volatile int16_t
    EchoBuffer[ECHO_BUFFER_SIZE];

volatile uint16_t
    EchoPointer = 0;

volatile uint8_t
    ClearEchoBuffer = TRUE;

//
// Data structure for timer configuration
//

TIM_HandleTypeDef
    Timer5_16Khz;

//
// Data structure for general purpose IO configuration
//

GPIO_InitTypeDef
    GpioInitStructure;

//
// Data structure for the D/A(DAC) Converter configuration
//

DAC_ChannelConfTypeDef
    DacInitStructure;

DAC_HandleTypeDef
    AudioDac;               // Structure for the audio digital to analog converter subsystem

//
// Data structures for the A/D Converter configuration
//

ADC_HandleTypeDef
    AudioAdc,
    ReferenceAdc,
    PitchShiftOffsetAdc;

volatile int
    ButtonCount = 0,
    ButtonState = RELEASED,
    Effect = NO_EFFECT;

//
// Buffering system variables
//
```

```
volatile int
    ADCPTR = 0;

volatile struct tBuffer
    Buffers[NUMBER_OF_BUFFERS];

volatile int
    WindowingState = 0,
    WindowingDone = FALSE;

//
// 4 times the size of the main buffer to compensate for addition of complex numbers and that we are
processing
// 2 buffers at a time
//

float
    delayedBuf[SIZE*4],
    procBuf[SIZE*4];

int
    AD_Offset;

void Init_GPIO_Port(uint32_t pin, uint32_t mode, uint32_t speed, uint32_t pull, char bus)
{
    GPIO_InitTypeDef GPIO_InitStructure; //a handle to initialize GPIO

    GPIO_InitStructure.Pin = pin;
    GPIO_InitStructure.Mode = mode;
    GPIO_InitStructure.Speed = speed;
    GPIO_InitStructure.Pull = pull;
    GPIO_InitStructure.Alternate = 0;
    if (bus == 'A') {
        HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);
    } else if (bus == 'B') {
        HAL_GPIO_Init(GPIOB, &GPIO_InitStructure);
    } else if (bus == 'C') {
        HAL_GPIO_Init(GPIOC, &GPIO_InitStructure);
    } else if (bus == 'D') {
        HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);
    } else if (bus == 'E') {
        HAL_GPIO_Init(GPIOE, &GPIO_InitStructure);
    }
}

void Init_GPIO_Port_Default_Speed_Pull(uint32_t pin, uint32_t mode, char bus)
{
    Init_GPIO_Port(pin, mode, GPIO_SPEED_MEDIUM, GPIO_NOPULL, bus);
}

// important for these to be in global as they need to be accessed in interrupt service routine
TIM_HandleTypeDef   DisplayTimer;
TIM_HandleTypeDef   LEDDisplayTimer;
TIM_HandleTypeDef   FrequencySpectrumGeneratorTimer;
void ConfigureTimers()
{
    __HAL_RCC_TIM3_CLK_ENABLE();
    DisplayTimer.Instance = TIM3;
    DisplayTimer.Init.Period = 49;//period & prescaler combination for 200 Hz frequency
    DisplayTimer.Init.Prescaler = 8399;
    DisplayTimer.Init.CounterMode = TIM_COUNTERMODE_UP;
    DisplayTimer.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_Base_Init( &DisplayTimer );

    HAL_NVIC_SetPriority( TIM3_IRQn, TIM3_PRIORITY, TIM3_PRIORITY);
    //set priority for the interrupt. Value 0 corresponds to highest priority
    HAL_NVIC_EnableIRQ( TIM3_IRQn );//Enable interrupt function request of Timer3

    __HAL_TIM_ENABLE_IT( &DisplayTimer, TIM_IT_UPDATE );// Enable timer interrupt flag to be set when
timer count is reached
    __HAL_TIM_ENABLE( &DisplayTimer );//Enable timer to start

    __HAL_RCC_TIM4_CLK_ENABLE();
    LEDDisplayTimer.Instance = TIM4;
```

```
    int prescaler = 105;
    LEDDisplayTimer.Init.Prescaler = prescaler - 1; // reduce to 800 kHz
    LEDDisplayTimer.Init.Period =  84000000 / prescaler / REFRESH_RATE / NUMBER_OF_LEDS - 1;
    // reduce to (REFRESH_RATE * NUMBER_OF_LEDS) frequency
    LEDDisplayTimer.Init.CounterMode = TIM_COUNTERMODE_UP;
    LEDDisplayTimer.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_Base_Init( &LEDDisplayTimer );

    HAL_NVIC_SetPriority( TIM4_IRQn, TIM4_PRIORITY, TIM4_PRIORITY);
    //set priority for the interrupt. Value 0 corresponds to highest priority
    HAL_NVIC_EnableIRQ( TIM4_IRQn );//Enable interrupt function request of Timer3

    __HAL_TIM_ENABLE_IT( &LEDDisplayTimer, TIM_IT_UPDATE );// Enable timer interrupt flag to be set when
timer count is reached
    __HAL_TIM_ENABLE( &LEDDisplayTimer );//Enable timer to start

    __HAL_RCC_TIM2_CLK_ENABLE();
    FrequencySpectrumGeneratorTimer.Instance = TIM2;
    prescaler = 140;
    FrequencySpectrumGeneratorTimer.Init.Period = prescaler - 1; // reduce to 600 kHz
    // reduce to (FRAMES_PER_SECOND * 2) frequency
    FrequencySpectrumGeneratorTimer.Init.Prescaler = 84000000 / prescaler / (FRAMES_PER_SECOND*2) - 1;
    FrequencySpectrumGeneratorTimer.Init.CounterMode = TIM_COUNTERMODE_UP;
    FrequencySpectrumGeneratorTimer.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_Base_Init( &FrequencySpectrumGeneratorTimer );

    HAL_NVIC_SetPriority( TIM2_IRQn, TIM2_PRIORITY, TIM2_PRIORITY);
    //set priority for the interrupt. Value 0 corresponds to highest priority
    HAL_NVIC_EnableIRQ( TIM2_IRQn );//Enable interrupt function request of Timer2

    __HAL_TIM_ENABLE_IT( &FrequencySpectrumGeneratorTimer, TIM_IT_UPDATE );// Enable timer interrupt flag
to be set when timer count is reached
    __HAL_TIM_ENABLE( &FrequencySpectrumGeneratorTimer );//Enable timer to start
}

void Configure_Ports()
{
    Init_GPIO_Port_Default_Speed_Pull(GPIO_PIN_12, GPIO_MODE_OUTPUT_PP, 'D');
    Init_GPIO_Port_Default_Speed_Pull(GPIO_PIN_0, GPIO_MODE_INPUT, 'A');

    Init_GPIO_Port_Default_Speed_Pull(BUTTON_1, GPIO_MODE_INPUT, 'B');
    Init_GPIO_Port_Default_Speed_Pull(BUTTON_2, GPIO_MODE_INPUT, 'C');
    Init_GPIO_Port_Default_Speed_Pull(BUTTON_3, GPIO_MODE_INPUT, 'B');
}

// utility inline functions to encapsulate the bus and port number of the row/col
inline void Write_Row_0 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOE, ROW_0, new_state); }
inline void Write_Row_1 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOE, ROW_1, new_state); }
inline void Write_Row_2 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOB, ROW_2, new_state); }
inline void Write_Row_3 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOD, ROW_3, new_state); }
inline void Write_Row_4 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOD, ROW_4, new_state); }
inline void Write_Row_5 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOD, ROW_5, new_state); }
inline void Write_Row_6 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOC, ROW_6, new_state); }
inline void Write_Row_7 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOC, ROW_7, new_state); }

inline void Write_Col_0 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOC, COL_0, new_state); }
inline void Write_Col_1 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOC, COL_1, new_state); }
inline void Write_Col_2 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOD, COL_2, new_state); }
inline void Write_Col_3 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOD, COL_3, new_state); }
inline void Write_Col_4 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOD, COL_4, new_state); }
inline void Write_Col_5 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOB, COL_5, new_state); }
inline void Write_Col_6 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOE, COL_6, new_state); }
inline void Write_Col_7 (uint16_t new_state) { HAL_GPIO_WritePin(GPIOE, COL_7, new_state); }

void Configure_LED_Display() {
    // init all rows and columns as output, medium speed, no pull
    Init_GPIO_Port_Default_Speed_Pull(ROW_0, GPIO_MODE_OUTPUT_PP, 'E');
    Init_GPIO_Port_Default_Speed_Pull(ROW_1, GPIO_MODE_OUTPUT_PP, 'E');
    Init_GPIO_Port_Default_Speed_Pull(ROW_2, GPIO_MODE_OUTPUT_PP, 'B');
    Init_GPIO_Port_Default_Speed_Pull(ROW_3, GPIO_MODE_OUTPUT_PP, 'D');
    Init_GPIO_Port_Default_Speed_Pull(ROW_4, GPIO_MODE_OUTPUT_PP, 'D');
    Init_GPIO_Port_Default_Speed_Pull(ROW_5, GPIO_MODE_OUTPUT_PP, 'D');
    Init_GPIO_Port_Default_Speed_Pull(ROW_6, GPIO_MODE_OUTPUT_PP, 'C');
    Init_GPIO_Port_Default_Speed_Pull(ROW_7, GPIO_MODE_OUTPUT_PP, 'C');
```

```
    Init_GPIO_Port_Default_Speed_Pull(COL_0, GPIO_MODE_OUTPUT_PP, 'C');
    Init_GPIO_Port_Default_Speed_Pull(COL_1, GPIO_MODE_OUTPUT_PP, 'C');
    Init_GPIO_Port_Default_Speed_Pull(COL_2, GPIO_MODE_OUTPUT_PP, 'D');
    Init_GPIO_Port_Default_Speed_Pull(COL_3, GPIO_MODE_OUTPUT_PP, 'D');
    Init_GPIO_Port_Default_Speed_Pull(COL_4, GPIO_MODE_OUTPUT_PP, 'D');
    Init_GPIO_Port_Default_Speed_Pull(COL_5, GPIO_MODE_OUTPUT_PP, 'B');
    Init_GPIO_Port_Default_Speed_Pull(COL_6, GPIO_MODE_OUTPUT_PP, 'E');
    Init_GPIO_Port_Default_Speed_Pull(COL_7, GPIO_MODE_OUTPUT_PP, 'E');

    // turn off all columns
    Write_Col_0(GPIO_PIN_RESET);
    Write_Col_1(GPIO_PIN_RESET);
    Write_Col_2(GPIO_PIN_RESET);
    Write_Col_3(GPIO_PIN_RESET);
    Write_Col_4(GPIO_PIN_RESET);
    Write_Col_5(GPIO_PIN_RESET);
    Write_Col_6(GPIO_PIN_RESET);
    Write_Col_7(GPIO_PIN_RESET);

    // turn off all rows
    Write_Row_0(GPIO_PIN_RESET);
    Write_Row_1(GPIO_PIN_RESET);
    Write_Row_2(GPIO_PIN_RESET);
    Write_Row_3(GPIO_PIN_RESET);
    Write_Row_4(GPIO_PIN_RESET);
    Write_Row_5(GPIO_PIN_RESET);
    Write_Row_6(GPIO_PIN_RESET);
    Write_Row_7(GPIO_PIN_RESET);

    Buffer_Init();
}

/**
 * Resets buffer head and tail for empty buffer
 */
inline void Buffer_Clear()
{
    buffer_head = 0;
    buffer_tail = -1; // flag that the buffer is empty
}

/**
 * Indicates if buffer is empty
 */
inline int Buffer_Is_Empty()
{
    return buffer_tail == -1;
}

/**
 * frame[] MUST have length NUM_OF_COLS
 * Returns 0 if buffer full, 1 if success
 */
char Buffer_Pushback(char frame[])
{
    if (buffer_tail == -1) {
        //buffer is empty, update actual available spot
        buffer_tail = 0;
    } else if (buffer_tail == buffer_head)
    {
        return 0; // buffer is full
    }

    // copy frame to buffer
    memcpy(display_buffer[buffer_tail], frame, NUM_OF_COLS);

    buffer_tail++; //increment to next available spot
    buffer_tail %= buffer_length; // wrap around to beginning of buffer

    return 1; //pushback success
}

/**
 * Pops front of buffer and copies to destination. If empty, nothing is copied.
 * dest[] MUST be length NUM_OF_COLS
```

```
 * Returns 0 if buffer empty, 1 if successfully copied
 */
char Buffer_Pop(char dest[])
{
    if (buffer_tail == -1)
    {
        return 0; //buffer is empty
    }

    // copy frame to buffer
    memcpy(dest, display_buffer[buffer_head], NUM_OF_COLS);

    buffer_head++;
    buffer_head %= buffer_length;

    if (buffer_head == buffer_tail) {
        //buffer is empty
        Buffer_Clear();
    }

    return 1;
}

/**
 * Initializes buffer with all 0's. Returns 1 when successful
 */
int Buffer_Init()
{
    char all_zeros[NUM_OF_COLS];
    for (int i = 0; i < NUM_OF_COLS; i++) {
        all_zeros[i] = 0;
    }

    for (int i = 0; i < buffer_length; i++) {
        memcpy(display_buffer[i], all_zeros, NUM_OF_COLS);
    }

    Buffer_Clear();

    return 1;
}

/**
 * turns on all LEDs for testing
 */
void Display_All_On() {
    for (int i = 0; i < NUM_OF_COLS; i++) {
        current_frame[i] = 0xFF;
    }
}

/**
 * turns off all LEDs for testing
 */
void Display_All_Off() {
    for (int i = 0; i < NUM_OF_COLS; i++) {
        current_frame[i] = 0;
    }
}

/**
 * If LED_Array is on, toggle off. If off, toggle on.
 */
void Toggle_Display_State() {
    if (LED_Array_State) {
        Display_All_Off();
        LED_Array_State = 0;
    } else {
        Display_All_On();
        LED_Array_State = 1;
    }
}

/**
 * Fill buffer with frames with one LED at a time, cycling through all LEDs
```

```
 * Precondition: buffer is at least framesToRepeat*64 long
 */
void Display_Scan_Across_LEDs() {
    const int framesToRepeat = 3;
    char frame[NUM_OF_COLS];

    for (int i = 0; i < NUM_OF_COLS; i++)
    {
        frame[i] = 0;
    }
    for (int currentLED = 0; currentLED < NUMBER_OF_LEDS; currentLED++)
    {
        int row = currentLED/NUM_OF_COLS;
        int col = currentLED%NUM_OF_COLS;

        frame[row] = 1 << col;

        for (int i = 0; i < framesToRepeat; i++)
        {
            Buffer_Pushback(frame);
        }

        frame[row] = 0;
    }

    for (int i = 0; i < NUM_OF_COLS; i++) {
        frame[i] = 0;
    }

    Buffer_Pushback(frame);
}

/**
 * Creates a bar of height 'height' in the specified column 'col'
 * Col must be <= NUM_OF_COLS
 */
void Create_Column_With_Height(char dest[], int col, int height) {
    char col_flag = 1 << col;
    for (int i = 1; i <= NUM_OF_COLS; i++)
    {
        if (i <= height) {
            dest[NUM_OF_COLS - i] = dest[NUM_OF_COLS - i] | col_flag; // force it to be 1
        } else {
            dest[NUM_OF_COLS - i] = dest[NUM_OF_COLS - i] & (0xFF ^ (col_flag)); // force it to be 0
        }
    }
}

/**
 * source is _source[frame number]
 * _source_length is the number of frames
 * message_length is the total number of columns in the message
 * direction is the direction the image pans
 */
void       Fill_Buffer_With_Panning_Image(int       _source_rows,       int       _source_cols,       char
source[_source_rows][_source_cols],
        int message_length, int direction) {
    char frame[NUM_OF_COLS];

//  char (*source)[_source_length] = _source;

    // Zero out the frame
    for (int i = 0; i < NUM_OF_COLS; i++) {
        frame[i] = 0;
    }

    // Start with blank frame
    Buffer_Pushback(frame);

    for (int current_index = 0; current_index < message_length; current_index++) {
        int source_index = current_index / NUM_OF_COLS;
        if (source_index >= _source_rows) { break; }
        int source_frame_index = current_index % NUM_OF_COLS;

        switch (direction) {
```

```
        case (LEFT_TO_RIGHT):
        case (RIGHT_TO_LEFT):
            for (int i = 0; i < NUM_OF_COLS; i++) {
                frame[i] = frame[i] << 1;

                char source_char = source[source_index][i];
                // truncate everything right of column
                char right_shifted = source_char >> (NUM_OF_COLS - 1 - source_frame_index);

                // remove everything left of column
                char right_col_only = right_shifted & (0xFE ^ 0xFF);

                // add in only the right column
                frame[i] = frame[i] | right_col_only;
            }
            break;
        case (BOTTOM_TO_TOP):
            for (int i = 0; i < NUM_OF_COLS - 1; i++) {
                // shift rows up by one
                frame[i] = frame[i + 1];
            }

            // add in new row at bottom
            frame[NUM_OF_COLS - 1] = source[source_index][source_frame_index];
            break;
//      case (RIGHT_TO_LEFT):
//          for (int i = 0; i < NUM_OF_COLS; i++) {
//              frame[i] = frame[i] >> 1;
//
//              char source_char = source[source_index][i];
//              // truncate everything left of column
//              char left_shifted = source_char << (NUM_OF_COLS - 1 - source_frame_index);
//
//              // remove everything right of column
//              char left_col_only = left_shifted & (0x7F ^ 0xFF);
//
//              // add in only the right column
//              frame[i] = frame[i] | left_col_only;
//          }
//          break;
        default:
            trace_printf("Not implemented exception. Invalid direction.");
            break;
        }

        Buffer_Pushback(frame);
        Buffer_Pushback(frame);
    }
}

/**
    111       111       111       111
  11   11   11   11   11   11   11   11
 1       1 1       1 1       1 1       1
1         1         1         1         1
 1       1 1       1 1       1 1       1
  11   11   11   11   11   11   11   11
    111       111       111       111

in hex:

0x0e    0x03    0x80    0xe0    0x38    0x00
0x31    0x8c    0x63    0x18    0xc6    0x00
0x40    0x50    0x14    0x05    0x01    0x00
0x80    0x20    0x08    0x02    0x00    0x80
0x40    0x50    0x14    0x05    0x01    0x00
0x31    0x8c    0x63    0x18    0xc6    0x00
0x0e    0x03    0x80    0xe0    0x38    0x00
0x00    0x00    0x00    0x00    0x00    0x00
 */
void Display_Sine_Wave() {
    char sine_wave[7][NUM_OF_COLS] = {
            { 0x0e, 0x31, 0x40, 0x80, 0x40, 0x31, 0x0e, 0x00 },
            { 0x03, 0x8c, 0x50, 0x20, 0x50, 0x8c, 0x03, 0x00 },
            { 0x80, 0x63, 0x14, 0x08, 0x14, 0x63, 0x80, 0x00 },
```

```
            { 0xe0, 0x18, 0x05, 0x02, 0x05, 0x18, 0xe0, 0x00 },
            { 0x38, 0xc6, 0x01, 0x00, 0x01, 0xc6, 0x38, 0x00 },
            { 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00 },
            { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } };
    Fill_Buffer_With_Panning_Image(7, NUM_OF_COLS, sine_wave, 49, LEFT_TO_RIGHT);
}

/**
   1     1       11          11       1     1
    1   1       1111        1111       1   1
   1111111     111111      111111     1111111
  11 111 11   11 11 11    11 11 11   11 111 11
 11111111111  11111111    11111111  11111111111
 1 1111111 1    1  1        1  1    1 1111111 1
 1 1     1 1   1 11 1      1 11 1   1 1     1 1
    11 11     1 1  1 1    1 1  1 1     11 11

in hex:

10  40  60  06  02  08
08  80  F0  0F  01  10
1F  C1  F8  1F  83  F8
37  63  6C  36  C6  EC
7F  F3  FC  3F  CF  FE
5F  D0  90  09  0B  FA
50  51  68  16  8A  0A
0D  82  94  29  41  B0
 */
void Invade_Space()
{
    char space_invaders[7][NUM_OF_COLS] = {
            { 0x10, 0x08, 0x1F, 0x37, 0x7F, 0x5F, 0x50, 0x0D },
            { 0x40, 0x80, 0xC1, 0x63, 0xF3, 0xD0, 0x51, 0x82 },
            { 0x60, 0xF0, 0xF8, 0x6C, 0xFC, 0x90, 0x68, 0x94 },
            { 0x06, 0x0F, 0x1F, 0x36, 0x3F, 0x09, 0x16, 0x29 },
            { 0x02, 0x01, 0x83, 0xC6, 0xCF, 0x0B, 0x8A, 0x41 },
            { 0x08, 0x10, 0xF8, 0xEC, 0xFE, 0xFA, 0x0A, 0xB0 },
            { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
    };
    Fill_Buffer_With_Panning_Image(7, NUM_OF_COLS, space_invaders, 55, RIGHT_TO_LEFT);
}

/**

   1111 111 11111 1111 1   1         1         1
   1  1  1    1   1    1   1  1     1 1       1
   1111  1    1   1    11111 1 1   1   1     1
   1     1    1   1    1   1    1 1     1   1
   1    111   1   1111 1   1     1       1 1
                                          1
111111111111111111111111111111111111111111111111

in hex:

00  00  00  00  00  00
1E  EF  BD  10  04  01
12  42  21  12  0A  02
1E  42  21  F5  11  04
10  42  21  10  A0  88
10  E2  3D  10  40  50
00  00  00  00  00  20
FF  FF  FF  FF  FF  FF
 */
void Display_Pitch_Shift()
{
    char pitch_shift_message[7][NUM_OF_COLS] = {
            { 0x00, 0x1E, 0x12, 0x1E, 0x10, 0x10, 0x00, 0xFF },
            { 0x00, 0xEF, 0x42, 0x42, 0x42, 0xE2, 0x00, 0xFF },
            { 0x00, 0xBD, 0x21, 0x21, 0x21, 0x3D, 0x00, 0xFF },
            { 0x00, 0x10, 0x12, 0xF5, 0x10, 0x10, 0x00, 0xFF },
            { 0x00, 0x04, 0x0A, 0x11, 0xA0, 0x40, 0x00, 0xFF },
            { 0x00, 0x01, 0x02, 0x04, 0x88, 0x50, 0x20, 0xFF },
            { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
    };
    Fill_Buffer_With_Panning_Image(7, NUM_OF_COLS, pitch_shift_message, 56, RIGHT_TO_LEFT);
```

```
}

/**
                                             1
   1111 1111 1   1 11111          1         1
   1    1    1   1 1   1   1     1 1       1
   1111 1    11111 1   1  1 1   1   1     1
   1    1    1   1 1   1     1 1     1   1
   1111 1111 1   1 11111      1       1 1
                                       1
111111111111111111111111111111111111111111111111

in hex:

00  00  00  00  00  04
1E  F4  5F  00  20  08
10  84  51  10  50  10
1E  87  D1  28  88  20
10  84  51  05  04  40
1E  F4  5F  02  02  80
00  00  00  00  01  00
FF  FF  FF  FF  FF  FF
 */
void Display_Echo()
{
    char echo_message[7][NUM_OF_COLS] = {
            { 0x00, 0x1E, 0x10, 0x1E, 0x10, 0x1E, 0x00, 0xFF },
            { 0x00, 0xF4, 0x84, 0x87, 0x84, 0xF4, 0x00, 0xFF },
            { 0x00, 0x5F, 0x51, 0xD1, 0x51, 0x5F, 0x00, 0xFF },
            { 0x00, 0x00, 0x10, 0x28, 0x05, 0x02, 0x00, 0xFF },
            { 0x00, 0x20, 0x50, 0x88, 0x04, 0x02, 0x01, 0xFF },
            { 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x00, 0xFF },
            { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
    };
    Fill_Buffer_With_Panning_Image(7, NUM_OF_COLS, echo_message, 56, RIGHT_TO_LEFT);
}

/**

  1111 111 11111 1111  1   1111 1111 1   1 11111
  1  1  1    1   1    1 1  1    1    1   1 1   1
  1111  1    1   1     1   1111 1    11111 1   1
  1     1    1   1    1 1  1    1    1   1 1   1
  1    111   1   1111  1   1111 1111 1   1 11111

111111111111111111111111111111111111111111111111

in hex:

00  00  00  00  00  00
3D  DF  79  1E  F4  5F
24  84  42  90  84  51
3C  84  41  1E  87  D1
20  84  42  90  84  51
21  C4  79  1E  F4  5F
00  00  00  00  00  00
FF  FF  FF  FF  FF  FF
 */
void Display_Pitch_Echo()
{
    char pitch_echo_message[7][NUM_OF_COLS] = {
            { 0x00, 0x3D, 0x24, 0x3C, 0x20, 0x21, 0x00, 0xFF },
            { 0x00, 0xDF, 0x84, 0x84, 0x84, 0xC4, 0x00, 0xFF },
            { 0x00, 0x79, 0x42, 0x41, 0x42, 0x79, 0x00, 0xFF },
            { 0x00, 0x1E, 0x90, 0x1E, 0x90, 0x1E, 0x00, 0xFF },
            { 0x00, 0xF4, 0x84, 0x87, 0x84, 0xF4, 0x00, 0xFF },
            { 0x00, 0x5F, 0x51, 0xD1, 0x51, 0x5F, 0x00, 0xFF },
            { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }
    };
    Fill_Buffer_With_Panning_Image(7, NUM_OF_COLS, pitch_echo_message, 56, RIGHT_TO_LEFT);
}

void Display_Debugging() {
    Display_Sine_Wave();
}
```

```
void Display_Mode() {
    Buffer_Clear();
    if (pitch_shift_state == ENABLE_PITCH_SHIFT && echo_state == ENABLE_ECHO) {
        Display_Pitch_Echo();
    } else if (pitch_shift_state) {
        Display_Pitch_Shift();
    } else if (echo_state) {
        Display_Echo();
    } else {
        Invade_Space();
    }
}

/**
 * Clears out echo buffer to AD_Offset so when it's subtracted it becomes 0
 */
inline void Echo_Buffer_Clear()
{
    for (int i = 0; i < ECHO_BUFFER_SIZE; i++)
    {
        EchoBuffer[i] = AD_Offset;
    }

    EchoPointer = 0;
}

/**
 * Appends value to end of echo buffer
 */
inline void Echo_Buffer_Pushback(int16_t value)
{
    EchoBuffer[EchoPointer] = value;

    // increment and wrap around
    EchoPointer++;
    EchoPointer %= ECHO_BUFFER_SIZE;
}

/**
 * Returns value of EchoBuffer[EchoPointer - 1], wrapping the index around to ECHO_BUFFER_SIZE
 */
inline int16_t Echo_Buffer_Pop()
{
    int index = EchoPointer + 1;
    index %= ECHO_BUFFER_SIZE;

    return EchoBuffer[index];
}

void Update_State()
{
    if (button_state_PB11) {
        previous_state_PB11 = 1;
    } else {
        if (previous_state_PB11) {
            //falling edge triggered
            pitch_shift_state = !(pitch_shift_state);

            Display_Mode();
        }
        previous_state_PB11 = 0;
    }

    if (button_state_PC4) {
        previous_state_PC4 = 1;
    } else {
        if (previous_state_PC4) {
            //falling edge triggered
            echo_state = !(echo_state);

            Display_Mode();

            if (echo_state != ENABLE_ECHO)
            {
```

```
                ClearEchoBuffer = TRUE;
            }
            else if (ClearEchoBuffer == TRUE)
            {
                // Zero out buffer
                Echo_Buffer_Clear();
                ClearEchoBuffer = FALSE;
            }
        }
        previous_state_PC4 = 0;
    }

    if (button_state_PB1) {
        previous_state_PB1 = 1;
    } else {
        if (previous_state_PB1) {
            //falling edge triggered

            Display_Mode();
        }
        previous_state_PB1 = 0;
    }
}

/**
 * Name: TIM5_IRQHandler
 *
 * Description: Time 5 interrupt service routine call 16,000 times a second.
 * Inputs:
 *      None
 *
 * Output:
 *      None
 *
 * Process:
 *
 *      Send audio signal to D/A converter
 *      Sample audio input
 *      Do echoing effect
 *      Handle windowing state update
 *      Update the LED display
 *      Detect button press and remove bounce
 *      Switch effects mode
 *
 */
void TIM5_IRQHandler(void)
{

    int16_t
        AudioSignal;

    TIMER_DEBUG_SIGNAL_ON;

//
// Check for timer update interrupt
//
    if ( __HAL_TIM_GET_FLAG( &Timer5_16Khz, TIM_IT_UPDATE ) != RESET )
    {

//
// Check for buffer full status
//
        if( 3 == Buffers[ANALOG_OUT_OFFSET].Full )
        {
//
// Output the Audio stream to the D/A converter
//
            DAC -> DHR12R1 = Buffers[ANALOG_OUT_OFFSET].Buf[Buffers[ANALOG_OUT_OFFSET].Head];

//
// Advanced the head pointer and check for end of buffer
//
            Buffers[ANALOG_OUT_OFFSET].Head++;      //increment head

            if( Buffers[ANALOG_OUT_OFFSET].Head >= SIZE)
```

```
            {
//
// Set the head pointer to the start of the buffer
// Reset the buffer full status
//
                Buffers[ANALOG_OUT_OFFSET].Head = 0;
                Buffers[ANALOG_OUT_OFFSET].Full = 0;
            }
        }

//
// Get values from adc and fill the buffer. when it is full reset the
// head pointer and set status to full then increment ALL buffers
// the & 0x03 is to loop the buffers back to 0 when they get to 4
// the << 3 is to increase the volume due to only being a 12b adc
//

//
// See if the buffer is not full
//
        if( 0  == Buffers[ADCPTR].Full)
        {

//
// Take a reading of the analog input pin and remove the offset signal
//
            AudioSignal = HAL_ADC_GetValue( &AudioAdc ) - AD_Offset;

//
// If enabled do the echo effect on the raw signal
//
            if ( echo_state == ENABLE_ECHO )
            {
                // pop from one index ahead of current EchoPointer
                // (which was the AudioSignal value one second ago)
                Buffers[ADCPTR].Buf[Buffers[ADCPTR].Head]   =   AudioSignal   *   (1-ECHO_DAMPING)   +
(Echo_Buffer_Pop() - AD_Offset) * ECHO_DAMPING;

                Echo_Buffer_Pushback(AudioSignal); // pushback current AudioSignal
            }
            else
            {

//
// No echo effect. just store the data in the buffer
//

                Buffers[ADCPTR].Buf[Buffers[ADCPTR].Head] = AudioSignal;
            }

//
// Update the head pointer
//
            Buffers[ADCPTR].Head++;

//
// See if the buffer is full
//
            if( Buffers[ADCPTR].Head >= SIZE )
            {

//
// If this statement returns true then the FFT portion of the code has failed.
//
                if (( FALSE == WindowingDone ) && ( 0 != WindowingState ))
                {
//
// Fatal error
//
                    while ( TRUE );
                }

//
// Advance to the next buffer
//
```

```
                Buffers[ADCPTR].Head = 0;       // Reset the head pointer
                Buffers[ADCPTR].Full = 1;       // Buffer Full = 1
                ADCPTR = ( ADCPTR + 1 ) & BUFFERS_MASK;

//
// changes the state for the overlapping windowing system
//
                switch( WindowingState )
                {
                    case 0:
                    {
                        WindowingState = 1;
                        WindowingDone = FALSE;
                        break;
                    }

                    case 1:
                    {
                        WindowingState = 2;
                        WindowingDone = FALSE;
                        break;
                    }

                    case 2:
                    {
                        WindowingState = 3;
                        WindowingDone = FALSE;
                        break;
                    }

                    case 3:
                    {
                        WindowingState = 4;
                        WindowingDone = FALSE;
                        break;
                    }

                    case 4:
                    {
                        WindowingState = 3;
                        WindowingDone = FALSE;
                        break;
                    }

                    default:
                    {

//
// Invalid state. Should not get here
//
                        while ( TRUE );
                        break;
                    }
                }
            }
        }

//
// Start another conversion
//
        HAL_ADC_Start( &AudioAdc );


//
// Clear the timer update interrupt flag
//
        __HAL_TIM_CLEAR_FLAG( &Timer5_16Khz, TIM_IT_UPDATE );

    }
    TIMER_DEBUG_SIGNAL_OFF;
}

/**
 * A FFT table utility function that shifts the buffer elements so buffer[i] = buffer[i-PitchOffset]
 * Starts at start_index, which is the highest index and stops before end_index
```

```
 * Clears all elements for last PitchOffset number of elements with 0's.
 */
inline void ShiftBufferElementsUp( float *Buffer, int start_index, int end_index, int PitchOffset)
{
    int PitchShift;

    // Start at highest index, start_index, and grab elements from smaller indices,
    // stopping before writing past end_index
    PitchShift = start_index;
    while ( PitchShift >= end_index + PitchOffset )
    {
        Buffer[PitchShift] = Buffer[PitchShift-PitchOffset];
        Buffer[PitchShift+1] = Buffer[(PitchShift+1)-PitchOffset];
        PitchShift -= 2;
    }

    // Clear the remaining (duplicated) portion of the table
    while ( PitchShift >= end_index )
    {
        Buffer[PitchShift] = 0;
        PitchShift--;
    }
}

/**
 * A FFT table utility function that shifts the buffer elements so buffer[i] = buffer[i+PitchOffset]
 * Starts at start_index, which is the lowest index and stops before end_index
 * Clears all elements for last PitchOffset number of elements with 0's.
 */
inline void ShiftBufferElementsDown ( float *Buffer, int start_index, int end_index, int PitchOffset)
{
    int PitchShift;

    // Start at lowest index, start_index, and grab elements from higher indices,
    // stopping before writing past end_index
    PitchShift = start_index;
    while ( PitchShift < ( end_index - PitchOffset ))
    {
        Buffer[PitchShift] = Buffer[PitchShift+PitchOffset];
        Buffer[PitchShift+1] = Buffer[(PitchShift+1)+PitchOffset];
        PitchShift += 2;
    }

    // Clear the remaining (duplicated) portion of the table
    while ( PitchShift < end_index )
    {
        Buffer[PitchShift] = 0;
        PitchShift++;
    }
}

void PitchShift( float *Buffer )
{
//
// Pitch Shift by 32 bins in the FFT table
// Each bin contains one complex number comprised of one real and one imaginary floating point number
//
    int PitchOffset = (pitch_shift_offset >= 0)? pitch_shift_offset * 2: pitch_shift_offset * -2;
    //between -32 and 32, take absolute value

    // The FFT table is 2048 in length
    const int FFT_table_size = 2048;

    // The lower half, the indices [0, 1024), corresponds to positive frequencies
    // The upper half, the indices [1024, 2048), corresponds to negative frequencies

    // Shift frequencies up effect
    if (pitch_shift_offset > 0)
    {
        // Shift the lower half of the FFT table up
        ShiftBufferElementsUp(Buffer, (FFT_table_size / 2 - 2), 0, PitchOffset);

        // Shift the upper half of the FFT table down
        ShiftBufferElementsDown(Buffer, FFT_table_size / 2, FFT_table_size, PitchOffset);
    }
```

```
    // Shift frequencies down effect
    if (pitch_shift_offset < 0)
    {
        // Shift the lower half of the FFT table down
        ShiftBufferElementsDown(Buffer, 0, FFT_table_size / 2, PitchOffset);

        // Shift the upper half of the FFT table up
        ShiftBufferElementsUp(Buffer, (FFT_table_size - 2), FFT_table_size / 2, PitchOffset);
    }
}

int ConvertPitchShiftOffset(void)
{
    int
        ADCResult;

//
// Start a conversion
//
    HAL_ADC_Start( &PitchShiftOffsetAdc );

//
// Wait for end of conversion
//
    HAL_ADC_PollForConversion( &PitchShiftOffsetAdc, HAL_MAX_DELAY );

//
// Get the 8 bit result
//
    ADCResult = HAL_ADC_GetValue( &PitchShiftOffsetAdc );

    return(ADCResult);
}

int
main(int argc, char* argv[])
{
    // At this stage the system clock should have already been configured
    // at high speed.

    unsigned int loop;

    HAL_Init();// initializing HAL drivers

    __GPIOA_CLK_ENABLE(); // enabling clock for port A
    __GPIOB_CLK_ENABLE(); // enabling clock for port B
    __GPIOC_CLK_ENABLE(); // enabling clock for port C
    __GPIOD_CLK_ENABLE(); // enabling clock for port D
    __GPIOE_CLK_ENABLE(); // enabling clock for port E

    for( loop = 0; loop < NUMBER_OF_BUFFERS; loop++)
    {
        Buffers[loop].Head = 0;
        Buffers[loop].Full = 0;
        memset( (_PTR)&Buffers[loop].Buf, 0, sizeof( Buffers[loop].Buf ));
    }

    InitSystemPeripherals();

    Configure_Ports();
    Configure_LED_Display();

    Display_All_Off();

//  Display_Scan_Across_LEDs();

    Display_Sine_Wave();

    // Start timers LAST to ensure that no interrupts based on timers will
    // trigger before initialization of board is complete
    ConfigureTimers();

    HAL_GPIO_WritePin( GPIOD, GPIO_PIN_12, 1); // Signal initialization is complete on on-board LED
```

```
    int previous_state_PA0 = 0;

//
//  Take an Offset reading to remove the DC offset from the analog reading
//  Source PC2 ( ADC_CHANNEL_12 )
//
    AD_Offset = ConvertReference();

    // Infinite loop
    while (1)
    {
        if (button_state_PA0) {
            previous_state_PA0 = 1;
        } else {
            if (previous_state_PA0) {
                //falling edge triggered
                Buffer_Clear();
                Display_Debugging();
            }
            previous_state_PA0 = 0;
        }

        Update_State();

        WindowingFFT();
    }
}

void TIM3_IRQHandler() //Timer3 interrupt function
{
    __HAL_TIM_CLEAR_FLAG( &DisplayTimer, TIM_IT_UPDATE );//clear flag status

    // This interrupt service routine is timer driven at 200 Hz
    // If the current reading is the same as the reading during the previous
    // interrupt, then the button state is reliable and we feed this to the rest
    // of the system

    // Check on board button
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0)) {
        // button is pressed.
        if (previous_button_reading_PA0) {
            // if this is consistent with previous reading, set state to 1
            button_state_PA0 = 1;
        }
        //update previous reading to current reading
        previous_button_reading_PA0 = 1;
    } else {
        // button is not pressed
        if (!previous_button_reading_PA0) {
            // if this is consistent with previous reading, set state to 0
            button_state_PA0 = 0;
        }
        //update previous reading to current reading
        previous_button_reading_PA0 = 0;
    }

    // Check PC1 button
    if (HAL_GPIO_ReadPin(GPIOB, BUTTON_1)) {
            // button is pressed.
        if (previous_button_reading_PB11) {
            // if this is consistent with previous reading, set state to 1
            button_state_PB11 = 1;
        }
        //update previous reading to current reading
        previous_button_reading_PB11 = 1;
    } else {
        // button is not pressed
        if (!previous_button_reading_PB11) {
            // if this is consistent with previous reading, set state to 0
            button_state_PB11 = 0;
        }
        //update previous reading to current reading
        previous_button_reading_PB11 = 0;
    }
```

```
    // Check PC4 button
    if (HAL_GPIO_ReadPin(GPIOC, BUTTON_2)) {
            // button is pressed.
        if (previous_button_reading_PC4) {
            // if this is consistent with previous reading, set state to 1
            button_state_PC4 = 1;
        }
        //update previous reading to current reading
        previous_button_reading_PC4 = 1;
    } else {
        // button is not pressed
        if (!previous_button_reading_PC4) {
            // if this is consistent with previous reading, set state to 0
            button_state_PC4 = 0;
        }
        //update previous reading to current reading
        previous_button_reading_PC4 = 0;
    }

    // Check PB1 button
    if (HAL_GPIO_ReadPin(GPIOB, BUTTON_3)) {
            // button is pressed.
        if (previous_button_reading_PB1) {
            // if this is consistent with previous reading, set state to 1
            button_state_PB1 = 1;
        }
        //update previous reading to current reading
        previous_button_reading_PB1 = 1;
    } else {
        // button is not pressed
        if (!previous_button_reading_PB1) {
            // if this is consistent with previous reading, set state to 0
            button_state_PB1 = 0;
        }
        //update previous reading to current reading
        previous_button_reading_PB1 = 0;
    }

    // Check potentiometer of pitch_shift_offset if ENABLE_PITCH_SHIFT
    if (pitch_shift_state == ENABLE_PITCH_SHIFT) {
        int pitch_shift_offset_raw = ConvertPitchShiftOffset(); // 0 to 255

        pitch_shift_offset_raw = 64.0/255 * pitch_shift_offset_raw; // reduce range to 0 to 64
        pitch_shift_offset -= 32; // shift range to -32 to 32;
    } else {
        pitch_shift_offset = 0;
    }
}

/**
 * WARNING: The LED array MUST be advanced from
 * increasing rows and columns
 */
void TIM4_IRQHandler() //Timer4 interrupt function
{
    __HAL_TIM_CLEAR_FLAG( &LEDDisplayTimer, TIM_IT_UPDATE ); //clear flag status

    if (current_row >= NUM_OF_COLS) {
        // at end of rows, need to advance to next column

        current_col++; //advance to next column
        current_row = 0; //restart row

        if (current_col >= NUM_OF_COLS) {
            // if the image has been displayed more than the number of times required
            // to achieve the desired REFRESH_RATE, pull the next image from buffer
            current_frame_number++;
            if (current_frame_number > times_to_repeat_frame) {
                Buffer_Pop(current_frame);

                current_frame_number = 0; //restart counting
            }
            current_col = 0; //restart column
        }
```

```
        // columns only need to be updated when the column number updates
        // for each case, turn off previous column, turn on current column
        switch(current_col) {
            case 0:
                Write_Col_7(GPIO_PIN_RESET);
                Write_Col_0(GPIO_PIN_SET);
                break;
            case 1:
                Write_Col_0(GPIO_PIN_RESET);
                Write_Col_1(GPIO_PIN_SET);
                break;
            case 2:
                Write_Col_1(GPIO_PIN_RESET);
                Write_Col_2(GPIO_PIN_SET);
                break;
            case 3:
                Write_Col_2(GPIO_PIN_RESET);
                Write_Col_3(GPIO_PIN_SET);
                break;
            case 4:
                Write_Col_3(GPIO_PIN_RESET);
                Write_Col_4(GPIO_PIN_SET);
                break;
            case 5:
                Write_Col_4(GPIO_PIN_RESET);
                Write_Col_5(GPIO_PIN_SET);
                break;
            case 6:
                Write_Col_5(GPIO_PIN_RESET);
                Write_Col_6(GPIO_PIN_SET);
                break;
            case 7:
                Write_Col_6(GPIO_PIN_RESET);
                Write_Col_7(GPIO_PIN_SET);
                break;
            default:
                //Should never enter this
                trace_printf("Invalid state in switch(current_col)");
                break;
        }
    }

    char enable_row = current_frame[current_col] & 1 << current_row;
    // for each case, turn off previous row, turn on current row
    switch(current_row) {
        case 0:
            if (enable_row) { Write_Row_0(GPIO_PIN_SET); }
            Write_Row_7(GPIO_PIN_RESET);
            break;
        case 1:
            if (enable_row) { Write_Row_1(GPIO_PIN_SET); }
            Write_Row_0(GPIO_PIN_RESET);
            break;
        case 2:
            if (enable_row) { Write_Row_2(GPIO_PIN_SET); }
            Write_Row_1(GPIO_PIN_RESET);
            break;
        case 3:
            if (enable_row) { Write_Row_3(GPIO_PIN_SET); }
            Write_Row_2(GPIO_PIN_RESET);
            break;
        case 4:
            if (enable_row) { Write_Row_4(GPIO_PIN_SET); }
            Write_Row_3(GPIO_PIN_RESET);
            break;
        case 5:
            if (enable_row) { Write_Row_5(GPIO_PIN_SET); }
            Write_Row_4(GPIO_PIN_RESET);
            break;
        case 6:
            if (enable_row) { Write_Row_6(GPIO_PIN_SET); }
            Write_Row_5(GPIO_PIN_RESET);
            break;
        case 7:
            if (enable_row) { Write_Row_7(GPIO_PIN_SET); }
```

```
            Write_Row_6(GPIO_PIN_RESET);
            break;
        default:
            //Should never enter this
            trace_printf("Invalid state in switch(current_row)");
            break;
    }

    current_row++; //move to next row
}

void TIM2_IRQHandler() //Timer2 interrupt function
{
    __HAL_TIM_CLEAR_FLAG( &FrequencySpectrumGeneratorTimer, TIM_IT_UPDATE );//clear flag status

    // This interrupt service routine is timer driven at 50 Hz

    // The FFT table is 2048 in length
    const int FFT_table_size = 2048;

    // Look at lower half of FFT table where higher indices correspond to higher frequencies
    // These indices are [0, 1023].
    // Each complex number takes up two elements in the float array. (One for real, one for imaginary)
    // We break 1024 elements, or 512 bins, into 8 groups, one for each column of the LED matrix
    // This means we investigate 512 / 8, or 64 bins, for each group
    // TODO  Since octaves are multiplicative, ideally we investigate in powers of 2
    // For each bin, we take max(real, imaginary) and add to the float. We are avoiding taking
    // the magnitude using sqrt(real^2 + imaginary^2) since sqrt is processor intensive and
    // we don't need the accuracy. max(real,imaginary) is an adequate approximation since
    // the max will dominate the square root anyway

    // group_sum / group_num_bins gives the average sort-of-magnitude in that group
    // average sort-of-magnitude / normalizing_constant brings the magnitude to a normalized_range
    // normalized_range * 8 gives the number of LEDs to light up in the column

    float group_sum = 0; // holds the accumulated sum for each group, used to average
    const int group_num_bins = 64; // 64 bins per group, which is converted to a column on the display

    const float normalizing_constant = 100; // divide the average by this, to normalize and convert to
bars

    int height_of_bar = 0; // the height to make the frequency bar
    char frequency_spectrum_frame[NUM_OF_COLS]; // to hold the frame being generated

    int bins_analyzed = 0; // the number of bins already analyzed in the group
    int current_col = 0; // tracking which column we are in

    if (Buffer_Is_Empty())
    {
        // Only generate the frequency spectrum frame if nothing is being displayed on LED display

        for (int i = 0; i < FFT_table_size / 2; i += 2)
        {
            // Add max(procBuf[i], procBuf[i+1]) to group_sum
            group_sum += (procBuf[i] > procBuf[i+1])? procBuf[i]: procBuf[i+1];
            bins_analyzed++;

            // if we have already analyzed the group, create a bar
            if (bins_analyzed >= group_num_bins * 2)
            {
                group_sum /= group_num_bins; //average magnitude

                group_sum /= normalizing_constant; // normalize

                height_of_bar = (int) (group_sum * 8); // calculate hight of bar

                Create_Column_With_Height(frequency_spectrum_frame, current_col, height_of_bar);
                // reset temporary variables
                bins_analyzed = 0;
                group_sum = 0;
                height_of_bar = 0;

                // increment to next column
                current_col++;
            }
```

```
        }
        Buffer_Pushback(frequency_spectrum_frame); // add it to buffer
    }
}

void InitSystemPeripherals( void )
{

    ADC_ChannelConfTypeDef
        sConfig;

//
// Enable device clocks TIMER and GPIO port E
//
    __HAL_RCC_TIM5_CLK_ENABLE();
    __HAL_RCC_GPIOE_CLK_ENABLE();
    __HAL_RCC_DAC_CLK_ENABLE();

//
// Enable ADC3 and GPIO port C clocks
//
    __HAL_RCC_ADC1_CLK_ENABLE();
    __HAL_RCC_ADC2_CLK_ENABLE();
    __HAL_RCC_ADC3_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOD_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();

//
// Enable GPIO Port E15 as an output ( used for timing with scope )
//

//  GpioInitStructure.Pin = GPIO_PIN_15 | GPIO_PIN_13;
//  GpioInitStructure.Mode = GPIO_MODE_OUTPUT_PP;
//  GpioInitStructure.Speed = GPIO_SPEED_FREQ_MEDIUM;
//  GpioInitStructure.Pull = GPIO_PULLUP;
//  GpioInitStructure.Alternate = 0;
//  HAL_GPIO_Init(GPIOD, &GpioInitStructure);

//
// Enable GPIO port A1 as an analog output
//
    GpioInitStructure.Pin = GPIO_PIN_4;
    GpioInitStructure.Mode = GPIO_MODE_ANALOG;
    GpioInitStructure.Speed = GPIO_SPEED_FREQ_MEDIUM;
    GpioInitStructure.Pull = GPIO_NOPULL;
    GpioInitStructure.Alternate = 0;
    HAL_GPIO_Init(GPIOA, &GpioInitStructure );

    EnableAudioCodecPassThru();

//
// Configure DAC channel 1
//
    AudioDac.Instance = DAC;

    HAL_DAC_Init( &AudioDac );

    DacInitStructure.DAC_Trigger = DAC_TRIGGER_NONE;
    DacInitStructure.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
    HAL_DAC_ConfigChannel( &AudioDac, &DacInitStructure ,DAC_CHANNEL_1 );

//
// Enable DAC channel 1
//
//
    HAL_DAC_Start( &AudioDac, DAC_CHANNEL_1 );

//
// Configure A/D converter channel 3
//

//
// Enable GPIO port C1, C2 and C5 as an analog input
```

```
//
    GpioInitStructure.Pin = GPIO_PIN_2 | GPIO_PIN_5; //GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_5;
    GpioInitStructure.Mode = GPIO_MODE_ANALOG;
    GpioInitStructure.Speed = GPIO_SPEED_FREQ_MEDIUM;
    GpioInitStructure.Pull = GPIO_NOPULL;
    GpioInitStructure.Alternate = 0;
    HAL_GPIO_Init(GPIOC, &GpioInitStructure );

//
// Configure audio A/D ( ADC2 ) for the audio stream
//

    AudioAdc.Instance = ADC2;
    AudioAdc.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV2;
    AudioAdc.Init.Resolution = ADC_RESOLUTION_12B;
    AudioAdc.Init.ScanConvMode = DISABLE;
    AudioAdc.Init.ContinuousConvMode = DISABLE;
    AudioAdc.Init.DiscontinuousConvMode = DISABLE;
    AudioAdc.Init.NbrOfDiscConversion = 0;
    AudioAdc.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T1_CC1;
    AudioAdc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    AudioAdc.Init.NbrOfConversion = 1;
    AudioAdc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    AudioAdc.Init.DMAContinuousRequests = DISABLE;
    AudioAdc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    HAL_ADC_Init( &AudioAdc );

//
// Select PORTC pin 5 ( ADC_CHANNEL_15 ) for the audio stream
//
    sConfig.Channel = ADC_CHANNEL_15;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_112CYCLES;
    sConfig.Offset = 0;

    HAL_ADC_ConfigChannel(&AudioAdc, &sConfig);
    HAL_ADC_Start( &AudioAdc );

//
// Configure level shifting reference A/D (ADC1)
//
    ReferenceAdc.Instance = ADC1;
    ReferenceAdc.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV2;
    ReferenceAdc.Init.Resolution = ADC_RESOLUTION_12B;
    ReferenceAdc.Init.ScanConvMode = DISABLE;
    ReferenceAdc.Init.ContinuousConvMode = DISABLE;
    ReferenceAdc.Init.DiscontinuousConvMode = DISABLE;
    ReferenceAdc.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T1_CC1;
    ReferenceAdc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    ReferenceAdc.Init.NbrOfConversion = 1;
    ReferenceAdc.Init.NbrOfDiscConversion = 0;
    ReferenceAdc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    ReferenceAdc.Init.DMAContinuousRequests = DISABLE;
    ReferenceAdc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    HAL_ADC_Init( &ReferenceAdc );
    HAL_ADC_Start( &ReferenceAdc );

    GpioInitStructure.Pin = GPIO_PIN_1;
    GpioInitStructure.Mode = GPIO_MODE_ANALOG;
    GpioInitStructure.Speed = GPIO_SPEED_FREQ_MEDIUM;
    GpioInitStructure.Pull = GPIO_NOPULL;
    GpioInitStructure.Alternate = 0;
    HAL_GPIO_Init(GPIOA, &GpioInitStructure );

//
// Configure pitch shift offset A/D (ADC3)
//
    PitchShiftOffsetAdc.Instance = ADC3;
    PitchShiftOffsetAdc.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV2;
    PitchShiftOffsetAdc.Init.Resolution = ADC_RESOLUTION_8B;
    PitchShiftOffsetAdc.Init.ScanConvMode = DISABLE;
    PitchShiftOffsetAdc.Init.ContinuousConvMode = DISABLE;
    PitchShiftOffsetAdc.Init.DiscontinuousConvMode = DISABLE;
    PitchShiftOffsetAdc.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T1_CC1;
    PitchShiftOffsetAdc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
```

```
    PitchShiftOffsetAdc.Init.NbrOfConversion = 1;
    PitchShiftOffsetAdc.Init.NbrOfDiscConversion = 0;
    PitchShiftOffsetAdc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    PitchShiftOffsetAdc.Init.DMAContinuousRequests = DISABLE;
    PitchShiftOffsetAdc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    HAL_ADC_Init( &PitchShiftOffsetAdc );
    HAL_ADC_Start( &PitchShiftOffsetAdc );

//
// Select PORTA pin 1 ( ADC_CHANNEL_1 ) for the pitch offset
//
    sConfig.Channel = ADC_CHANNEL_1;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_112CYCLES;
    sConfig.Offset = 0;

    HAL_ADC_ConfigChannel(&PitchShiftOffsetAdc, &sConfig);
    HAL_ADC_Start( &PitchShiftOffsetAdc );

//
// Initialize timer to 16Khz
//
    Timer5_16Khz.Instance = TIM5;
    Timer5_16Khz.Init.CounterMode = TIM_COUNTERMODE_UP;
    Timer5_16Khz.Init.Period = 250;
    Timer5_16Khz.Init.Prescaler = 20;
    Timer5_16Khz.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_Base_Init( &Timer5_16Khz );

//
// Enable the timer interrupt
//
    HAL_NVIC_SetPriority( TIM5_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ( TIM5_IRQn );

    __HAL_TIM_ENABLE_IT( &Timer5_16Khz, TIM_IT_UPDATE );

//
// Enable timer 5 update interrupt
//
    __HAL_TIM_ENABLE( &Timer5_16Khz );

}

int ConvertAudio(void)
{
    int
        ADCResult;

//
// Start a conversion
//
    HAL_ADC_Start( &AudioAdc );

//
// Wait for end of conversion
//
    HAL_ADC_PollForConversion( &AudioAdc, HAL_MAX_DELAY );

//
// Get the 12 bit result
//
    ADCResult = HAL_ADC_GetValue( &AudioAdc );

    return(ADCResult);
}

int ConvertReference(void)
{
    int
        ADCResult;

    ADC_ChannelConfTypeDef sConfig;

//
```

```
// Select the channel to convert and start the conversion
//
    sConfig.Channel = ADC_CHANNEL_12;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_112CYCLES;
    sConfig.Offset = 0;

    HAL_ADC_ConfigChannel(&ReferenceAdc, &sConfig);

//
// Start a conversion
//
    HAL_ADC_Start( &ReferenceAdc );

//
// Wait for end of conversion
//
    HAL_ADC_PollForConversion( &ReferenceAdc, HAL_MAX_DELAY );

//
// Get the 12 bit result
//
    ADCResult = HAL_ADC_GetValue( &ReferenceAdc );

    return(ADCResult);
}


#pragma GCC diagnostic pop

// ---------------------------------------------------------------------------
```