ELECTRONIC DRUM MACHINE (E.D.M)

by

Nyssa Backes

Senior Project

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

TABLE OF CONTENTS

LISTS OF TABLES AND FIGURES

Acknowledgements

Without the help of the amazing people of the 909 Project and Mickey Delp, of Delptronics, whose work I was allowed to build upon and modify, this project would not have been possible. Furthermore, several programs constructed for the MSP432 were based upon programs created with my lab partner, Andrea Levy, during CPE 329 several quarters ago, so her contributions must too be acknowledged with extreme gratitude. Finally, I am deeply indebted to my sister, Tessa, for her near-infinite knowledge of drums and her willingness to provide recording for my software-generated drum samples.

Additionally, without the invaluable help of my advisor, Dr. Joseph Callenes-Sloan, my project would be no more than a disorganized idea.

As always, much appreciation is due to my family, friends, and roommates for being a constant source of encouragement and catharsis.

Abstract

Many people argue that a drummer plays the most important role in a band of any genre. During live performances, the other band members rely on the drummer to keep the song moving forward at a proper and steady pace. Even in recorded music, drums help listeners sing and dance along with their favorite songs. Thus, achieving success as a band absolutely requires a drummer. So, if a band does not have a drummer, and cannot afford the hefty cost of hiring a session drummer, a touring drummer, or both, how can they possibly overcome this obstacle? Simple. Purchase an electronic drum machine, like EDM, which provides the rhythmic functionality of a human drummer at a fraction of the cost.

Essentially, the EDM is a small device that can produce 12 standard drum set based sounds individually (namely. crash cymbal, open high-hat, closed high-hat, tom hits, among others). These individual sounds can then play separately or chained together at any tempo and in any order, meaning users can program their own original drum beats. Furthermore, E.D.M. features a catalog of standard drum instrumentations which also play at any user defined tempo. For increased usability, the EDM supports multiple audio outputs and utilizes battery power efficiently.

I: Introduction

Drummers serve one important function in any band of any genre. Yet, for bands that lack a drummer, many financial obstacles stand in the way of acquiring one. Studio and touring drummers can charge as much as $400 per day for their services and simply purchasing a drum kit often represents a hefty financial investment as, "…professional drum kits can cost upwards of $10,000," [1]. So, if a band does not have a drummer, and cannot afford the substantial cost of hiring a session drummer, a touring drummer, or both, they can overcome this obstacle by purchasing an electronic drum machine, like E.D.M.

Popular music has employed electronic drum machines to bridge this gap for more than 50 years, as the first "drum machine" as we know them today, debuted in 1959 [2]. Yet, while mainstream artists like Sly Stone, Peter Gabriel, and the Bee Gees quickly adopted the technology to revolutionize their sound as early as the 1970s, electronic drum machines did not gain popularity until Roland released the revolutionary TR-808, colloquially called the 808, in 1980 [3]. As stated on the device's Wikipedia page, more, "…hit records," feature the 808 than "…any other drum machine,' [3]. Additionally, Wikipedia claims that the TR-808 has "…an influence on popular music comparable to, "…the Fender Stratocaster's influence on rock." Thus, from 1980 onward, electronic drum machines became far more than a quick fix for bands without drummers.

The TR-808 revolutionized the music scene in that it provided a user friendly and low-cost alternative to the drum machines that dominated the market at the time [4]. Furthermore, the following statement best summarizes the Roland TR-808's ubiquity and influence,

> "The 808 broke down the walls between genres, and spawned collaborations between some of the biggest acts from different spaces. Because the 808 was so adaptable, it was like the first open-sourced sound, with artists building on each other's interpretations and making it their own… The 808 [became] like the not-so-secret sauce of hit records…," [4].

Yet, while the 808 does not lack in prestige and preeminence, it does lack in updateability. The 808 is an analog machine, meaning that all 12 of its on-board drum sounds are hardware-generated [3]. This makes it extremely difficult for users to add new drum sounds to their 808, as doing so requires adding new circuitry to their machines. Aside from the obvious limitations of the space inside of the physical system that must accommodate such additions, the act of integrating new circuitry to an existing system is far beyond what most users are comfortable doing. This exact deficiency led to the rise of the 808's closest competitor, the totally digital Linn LM-1 Drum Computer [3].However, while updating the Linn LM-1 involved simple software changes, the LM-1's drum sounds could not be triggered by MIDI, which severely limits usability. Thus, the E.D.M. attempts to supplant both the 808 and the Linn LM-1 by providing the same basic functionality, with the additional characteristics of updateability and MIDI-triggering.

Much like the 808 and the Linn LM-1, the EDM provides all the rhythmic functionality of a human drummer, at a discounted cost. The EDM produces 12 standard drum set based sounds individually (namely crash cymbal, open high-hat, closed high-hat, tom hits, and others). These sounds may then play separately or chained together at any tempo and in any order, meaning that users can program their own original drum tracks. Furthermore, E.D.M. features a catalog of standard drum instrumentations, which also play at any user defined tempo. For increased usability, the EDM supports multiple audio outputs and utilizes robust battery power. Additionally, as the E.D.M's drum sounds are generated through a combination of analog and digital hardware, simple software updates can expand the on-board libraries of pre-programmed drum tracks and individual drum sounds. Thus, the E.D,M takes everything that made the Roland TR-808 great (inexpensive price tag, pre-programmed individual sounds and tracks, and a simple user interface) and makes it better by making it the entire system easily updateable.

II: Customer Needs, Requirements, and Specifications

Customer Needs Assessment

Identifying the individual groups comprising potential E.D.M. customers took place before determining any customer needs. I determined the potential customers' identity through solitary brainstorming and in-class group discussion. After such discussion, I employed the following methods to gather a thorough understanding of potential EDM customers' needs: researching existing competitive products, interviewing identified customers, logically inferring what a customer might desire from the product, and observing customers using similar products to meet their current needs [5]. Through such research, I discovered that, principally, potential customers need a lightweight and portable product, since many potential customers are either traveling musicians or aspiring music producers [5]. Additionally, accommodating a variety of musical genres maximizes the drum machine's customer satisfaction since it allows both musicians and producers creative freedom to experiment with many musical styles using only one piece of drum equipment. Lastly, since many potential E.D.M. customers are not drummers themselves, but rather musicians or producers mitigating the cost and situational difficulty of acquiring a human drummer, the E.D.M. needs a simple user interface so even drumming novices can understand and operate the device effectively.

Requirements and Specifications

Determining a broad list of all potential customer needs lent itself to easily determining the E.D.M.'s necessary marketing requirements. Essentially, the customers' purported needs, such as portability, versatility amongst genres, and others, translated directly into abstract marketing requirement statements. These requirements were then further decomposed into engineering specifications, again through a combination of solitary work and group analysis and discussion. Engineering requirements should provide concrete statements of how the E.D.M.'s functionalities meet the determined customer needs without providing specific solutions regarding how the E.D.M. technically implements the desired functionalities. Thus, specifications like "…supports interfacing with other electronic instruments," and "…operates using only battery power for at least 5 hours," describe how the E.D.M.'s functionalities meet the customer needs of portability and musical versatility, but do not specify exactly how, as such an over-specified description unnecessarily limits design space. The following table (TABLE I: E.D.M. Requirements and specifications) presents a summarized and synthesized list of the marketing requirements and engineering specifications determined through such analysis as previously described.
Additionally, configuring the EDM's memory, to store all pre-programmed and custom drum tracks, poses the most difficulty. Specifically in regards to the individual drum samples, compromises between sample quality and sample size must be made as high quality samples can be quite large and larger samples reduce the amount of pre-programmed and custom drum tracks that the E.D.M. can store. Furthermore, the sound quality of the hardware generated sounds must be considered when making such compromises, as a large disparity in quality between the hardware and software generated drum sounds only serves to degrade the sound quality of the resultant custom drum tracks. Additionally, the fundamental design choice with regard to memory implementation revolves around whether the MSP432 has enough on-board memory to accommodate the E.D.M's needs, or if off-board memory, potentially in the form of serial EEPROM, is required.

TABLE I
E.D.M. requirements and specifications

Marketing Requirements	Engineering Specifications	Justification
1	EDM stores at least 30 user-programmed tracks for playback.	Allowing users to program 30 individual tracks provides musicians the ability to program their drum tracks for an average set list at once.
1	Users program at least 30 individual tracks through interaction with the physical EDM system.	Users input user-programmed tracks via buttons and knobs on the front of the physical EDM unit.

1	EDM MIDI output volume ranges from 1 to 127 (maximum MIDI volume range).	Integers from 1 to 127 represent volume in MIDI terms. EDM produces a MIDI signal using any possible MIDI volume, thus maximizing sound quality.
1, 2, 3	EDM supports interfacing with both computers and other electronic instruments.	By interfacing with computers and other electronic instruments, the EDM can become an integrated part of a studio set-up or live stage show. Additionally, usability and sound quality increase as this allows additional sound filtering and processing outside of the EDM itself. Lastly, as the EDM can interface with computers, updates may be employed to add new libraries of drum sounds and pre-programmed tracks, thus allowing users to play a larger variety of musical styles and genres.
1, 2	EDM includes MIDI and 3.5 mm jack outputs.	Multiple audio outputs increases the EDM's usability, since it can interface with more devices, and increases sound quality by providing the opportunity for additional output filtering and synthesizing.
1, 6	EDM complies with all parts of ISO 97.00.20 standard regarding musical instruments and audio-visual equipment used for entertainment.	Compliance with this standard ensures the production of a safe and high-quality product.
2	EDM stores at least 20 pre-programmed tracks for playback.	Users who do not have any prior drumming ability may also use the EDM by selecting from 20 pre-programmed drum tracks.
2	EDM plays both pre-programmed and user-defined tracks over a time signature range of 1-128 beats per patterns, within 1% accuracy, and a tempo range of 20-255 bpm (beats per minute) within 1% accuracy.	Regardless of genre, most western music has a time signature in the range of 1-128 beats per pattern and a tempo in the range of 20-255 bpm (beats per minute). Thus, EDM accommodates the time signatures and tempos of most western music.
2	EDM displays the name of the track currently playing in a text format.	Displaying the name of the drum track currently playing makes the EDM easy to use as users can see which track they have selected for playback.
2, 4	EDM features dual power supplies, meaning batteries or AC wall power may be used.	While the EDM may operate using only battery power, it may also operate when plugged into a wall, which increases EDM's usability.
2, 3	EDM supports the uploading of new libraries of pre-programmed drum tracks and individual drum sounds via computer.	Adding new libraries of pre-programmed tracks and individual drum noises increases the variety of styles and genres that the EDM can accommodate. Furthermore, adding new libraries with a computer makes these updates user friendly and easily accessible.
2, 3	EDM includes a pre-programmed library of at least 11 individual drum-set based sounds.	Producing a variety of individual drum sounds lets users create their own drum patterns, beyond those included in the EDM. This increases the genres the EDM may suit, and lets the EDM better suit individual customers' needs

		as users can play a wider variety of musical styles and genres. Additionally, the inclusion of additional libraries allows for computer-based system updating.
4	EDM costs less than $150.00.	This price point places the EDM far below the average market value for comparable products.
4, 5	EDM operates using only battery power for at least 5 hours.	Operating only with battery power increases the portability and usability of the EDM. Additionally, having at least 5 hours of battery power affords an opportunity to perform a complete concert, approximately, using only battery power.
6	EDM still functions as required after a 2 foot drop onto a carpeted surface.	This simple durability standard ensures the EDM is sufficient for the demands of studio and touring musicians.
6	EDM features a metal chassis.	By including a metal chassis, the EDM becomes significantly more durable and far more resistant to drops and falls.
Marketing Requirements 1. Provide good sound quality. 2. Usable by performers of a variety of genres and experience levels. 3. Updateable. 4. Low cost. 5. Small, lightweight, and portable. 6. Durable		

The requirements and specifications table format derives from [6], Chapter 3.

III: Design

Functional Decomposition (Level 0 and 1)

Essentially, Figure 1 provides an overall functional decomposition of the EDM at the highest system level, level 0. This level simply shows the most abstract inputs and outputs to the overall EDM system. Additionally, Table III provides a more detailed analysis, and concise list of data types and informational content represented by each input and output signal shown in Figure 1.

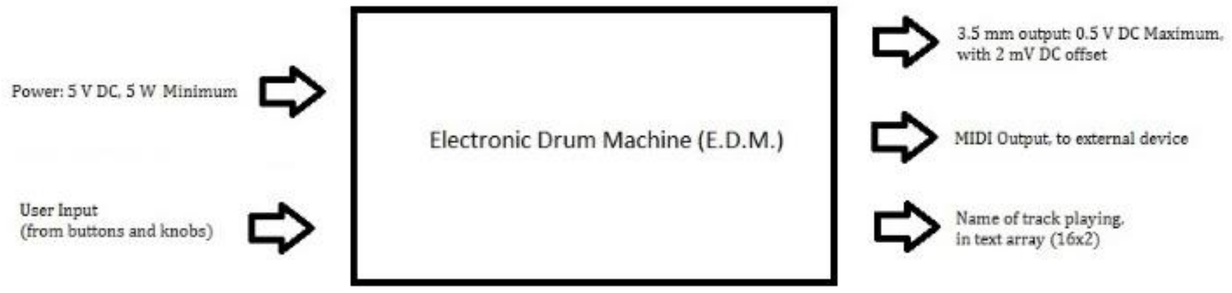

Power: 5 V DC, 5 W Minimum
User Input (from buttons and knobs)
Electronic Drum Machine (E.D.M.)
3.5 mm output: 0.5 V DC Maximum, with 2 mV DC offset
MIDI Output, to external device
Name of track playing, in text array (16x2)

Figure 1 – Level 0 Block Diagram

TABLE II
E.D.M. Level 0 Functional Requirements

Module	Electronic Drum Machine
Inputs	• Power: 5.5 V DC, 5 W minimum power • User Input, from buttons and knobs o Pre-programmed track selection o Individual drum noise selection ▪ Snare Drum ▪ Bass Drum ▪ High Tom ▪ Low Tom ▪ High-hat Open ▪ High-hat Closed ▪ High-hat with foot ▪ Crash cymbal ▪ Ride cymbal ▪ Ride cymbal bell ▪ Rim click o Tempo control selection (20-255 bpm, 5 bpm step) o Time signature control selection (1-128 beats per pattern, 1 beat per pattern step) o Volume Control
Outputs	• Audio output signal: 0.5 V DC, with 2 mV DC offset • MIDI output, to an external device • Name of track playing in text array (2 rows of 16 characters each)
Functionality	Take user input, from buttons and knobs on the EDM unit, and output corresponding drum track at user-specified tempo and time signature. Selected track outputs as a voltage on a 3.5 mm jack and as a MIDI signal. Also, name of output track displays in text on physical system.

The following diagram (see Fig. 2) provides a more detailed view into the modular structure of the E.D.M. and shows how information signals flow through the system to perform the desired actions. Additionally, the following tables (see Tables III - VIII) provide further explanation as to the specific inputs, outputs, and purpose of each inner block. This level 1 block diagram derives, in part, from the 9090 Project's documentation [7].

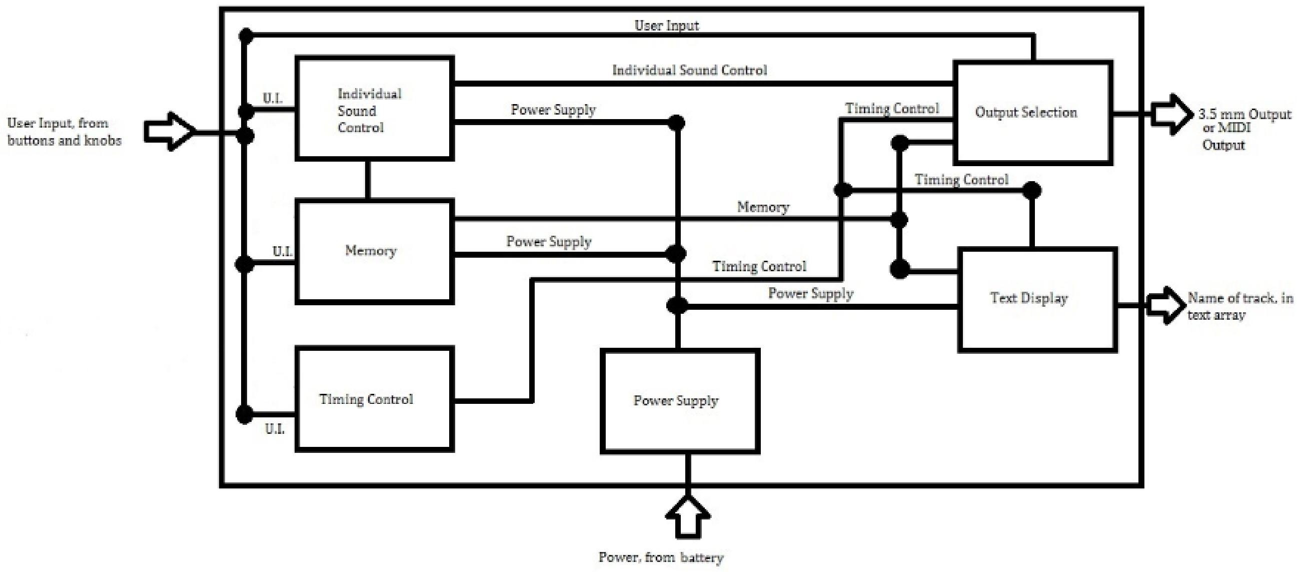


Figure 2 - Level 1 Block Diagram

TABLE III
E.D.M. Level 1 - Individual Drum Sound Control Functional Requirements

Module	Individual Drum Sound Control
Inputs	• User Input, from buttons and knobs • Users select which individual drum sound to play • Users also program custom drum tracks to memory by selecting individual drum sounds in the desired order. • Power o This module receives DC voltages, from the power supply module.
Outputs	• Memory o Programming custom drum tracks requires the individual drum sounds to be written to memory, in the desired order. • Output Selection o Individual drum sounds are output as either a MIDI signal or 3.5 mm signal.
Functionality	Overall, this module allows user to select individual drum sounds or to select drum sounds in a particular order so that they may be saved to memory as a custom drum track.

TABLE IV
E.D.M. Level 1 - Memory Functional Requirements

Module	Memory
Inputs	• User Input, from buttons and knobs o Users can select from their own custom drum tracks or the E.D.M.'s pre-programmed tracks, both of which are stored in the system's memory.
Outputs	• Text Display o Since the name of the track currently playing displays in text on the E.D.M.'s user interface, the name of the selected track must be output from memory. • Output Selection

	• The selected track must output from memory to the output selection module so that it may output from E.D.M.
Functionality	Overall, this module serves as the entire memory of the E.D.M. When users define custom drum tracks, all the necessary information (namely order of sounds and timing selection) are stored in the system's memory. Additionally, all the pre-programmed drum tracks are stored here. The system memory then transmits the selected track to the output.

TABLE V
E.D.M. Level 1 - Timing Control Functional Requirements

Module	Timing Control
Inputs	• User Input, from buttons and knobs o Users select desired time signature and tempo for the output track.
Outputs	• Output Selection o Information regarding the timing of the output track accompanies the output track in the output selection module.
Functionality	This module controls the timing of the output track. Users select from a range of tempos and time signatures for either a pre-programmed or a custom, user-defined track.

TABLE VI
E.D.M. Level 1 - Power Supply Functional Requirements

Module	Power Supply
Inputs	• Power, from battery or AC wall power o The E.D.M. system features dual power supplies, meaning AC wall power or batteries may supply power.
Outputs	• Individual Sound Control o The power supply provides DC voltages to the Individual Sound Control module. • Memory o The power supply provides DC voltages to the Memory module. • Text Display • The power supply provides DC voltages to the Text Display module
Functionality	The Power Supply module takes in battery power or AC wall power and provides DC voltages to the Individual Sound Control, Memory, and Text Display modules.

TABLE VII
E.D.M. Level 1 - Output Selection Functional Requirements

Module	Output Selection
Inputs	• User Input, from buttons and knobs o Users select from either 3.5 mm type output or MIDI type output. • Individual Sound Control o Individual drum sounds output as 3.5 mm signals or as MIDI signals. • Timing Control o Information regarding the user-selected tempo and time signature of the output are needed to output the proper track at the proper tempo and time signature.

Outputs	• 3.5 mm Output ○ Selected tracks and individual drum sounds output as a 3.5 mm signal. • MIDI Output ○ Selected tracks and individual drum sounds output as a MIDI signal.
Functionality	Users can select either 3.5 mm output or MIDI output format by using the E.D.M.'s buttons and knobs for input. Additionally, information regarding the selected timing of the track, as well as the selected track or desired individual drum sounds, pass into the output selection module.

TABLE VIII
E.D.M. Level 1 - Text Display Functional Requirements

Module	Text Display
Inputs	• Memory ○ The name of the track currently playing comes from the memory module. • Power Supply ○ The text array requires DC voltages from the power supply.
Outputs	• Name of track, in text array ○ The name of the track currently playing displays on the physical E.D.M. system in a text array.
Functionality	This module require DC voltages from the power supply and the name of the selected track from memory. This name displays in a text array on the E.D.M.

The diagrams shown below (see Figs. 3 and 4) provide a more detailed functional decomposition of two subsystems of the E.D.M. that remain ambiguous after the Level 1 decomposition. These subsystems, the individual sound control block and the output selection block, are thoroughly decomposed and described in Tables IX and X below.

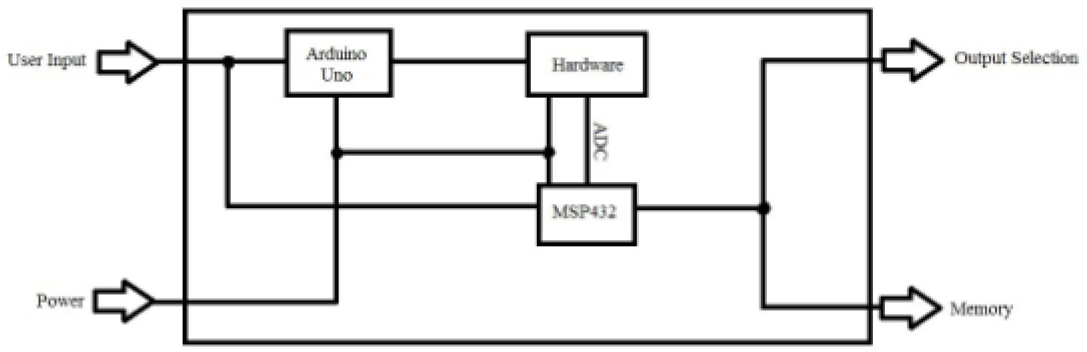


Figure 3 - Level 2 Block Diagram of Individual Sound Control Block

TABLE IX
E.D.M. Level 2 - Individual Sound Control Functional Requirements

Module	Individual Sound Control
Inputs	• User Input ○ Users select individual drum sounds using the E.D.M's external buttons and knobs. • Power Supply

	○ The hardware circuits, Arduino Uno, and MSP43 all require DC voltages to operate.
Outputs	• Output Selection ○ Individual drum sounds are output in MIDI or 3.5 mm. • Memory ○ Individual drum sounds are combined into user-defined patterns and saved in system memory.
Functionality	5 of the E.D.M's 11 individual drum sounds are generated by hardware, which is driven by an Arduino. The other 6 drum sounds are contained in software samples in the flash memory of the MSP432. When the sounds are selected they are then combined and processed in the MSP432 before being output or saved to memory.

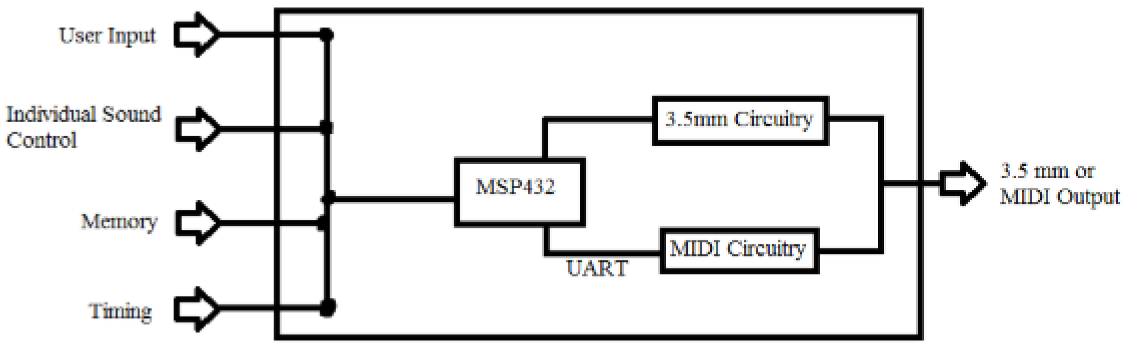


Figure 4 - Level 2 Block Diagram of Output Selection Block

TABLE X
E.D.M. Level 2 - Output Selection Functional Requirements

Module	Output Selection
Inputs	• Individual Sound Control ○ Individual drum sounds may be output as either 3.5 mm or MIDI. • Memory ○ User-defined or pre-programmed tracks may also be output as either 3.5 mm or MIDI. • Timing ○ Timing information, as well as signal content, must be output. • User Input ○ Users can directly select the desired output form.
Outputs	• 3.5mm Output ○ Individual drum sound, user-defined, or pre-programmed tracks may be output in 3.5 mm form. • MIDI Output ○ Individual drum sound, user-defined, or pre-programmed tracks may be output in MIDI form.
Functionality	The MSP432 takes input from the user, individual sound control, timing module, and memory module and then outputs the result as either a 3.5 mm signal or MIDI.

The functional requirements table format derives from [6], Chapter 5.

Technology Choices

Microcontroller Selection

While initially a field-programmable gate array (FPGA) was considered to serve as the main processing agent of the E.D.M., meaning that all of the logic used to combine and save the various drum patterns would occur in the FPGA. However, after some initial research and consideration a microcontroller, specifically the MSP432, was selected to serve as the main processing agent for a variety of reasons. Principally, the ubiquity of well documented examples and existing open-source projects involving the MSP432, or its nearly identical predecessor the MSP430, informed this decision as the same breadth of information regarding FPGAs was not as readily available. Additionally, the cost of the MSP432 was far less than that of comparable FPGAs that were considered, and I have significant personal experience with the MSP432 as I have completed an entire course at Cal Poly on its properties and applications.
After selecting the MSP42 as the main processing engine of the E.D.M. it was decided that an Arduino Uno should be employed to produce the square waves and white noise pulses needed to excite the hardware circuits into operation. This decision was principally motivated by the inexpensive and ubiquitous nature of the Arduino Uno, as well as a desire to free up program memory in the MSP432 to streamline drum track processing.

Hardware Generated Sounds

While the E.D.M. seeks to improve upon classic drum machines, like the LM-1 and the TR-808, it must also pay homage to those classics that have come before it to retain marketability toward fans of classical drum machines. In this way, physical hardware was chosen to produce the snare, bass, low tom, high tom, and rim click sounds as hardware provides a unique character to these sounds that has become canonically associated with such classic machines as the TR-808. The specific circuit configuration used derives from Mickey Delp's LDB-1, a project that sought to recreate the classic TR-808 sounds, with more compact and modern hardware than was available at the TR-808's inception in 1989 [8]. Further details of each individual circuit follow below. An Arduino Uno was employed to produce the square waves and white noise pulses used to excite the various hardware circuits into operation. Arduino code was composed so that, upon individual button presses, each individual hardware sound would effectively be "turned on" until the button was pressed once again. In reality, once users press a button, the individual circuit corresponding to that button is excited into operation, and then the MSP432 samples and stores data about each sound using 14-bit analog-digital-conversion (ADC). It is important to note that this sampling only occurs once, regardless of how long the user allows the button controlling circuit function to remain "on." As this data is stored in the MSP432's volatile memory, the information is erased each time the E.D.M. is powered off, thus ensuring a new "recording" is constructed every time the E.D.M. is powered on. In this way, the E.D.M. seeks to recreate the hardware-based non-uniformities in sound that helped to make drum machines like the TR-808 and TR-909 sound distinct and original.

Snare Drum

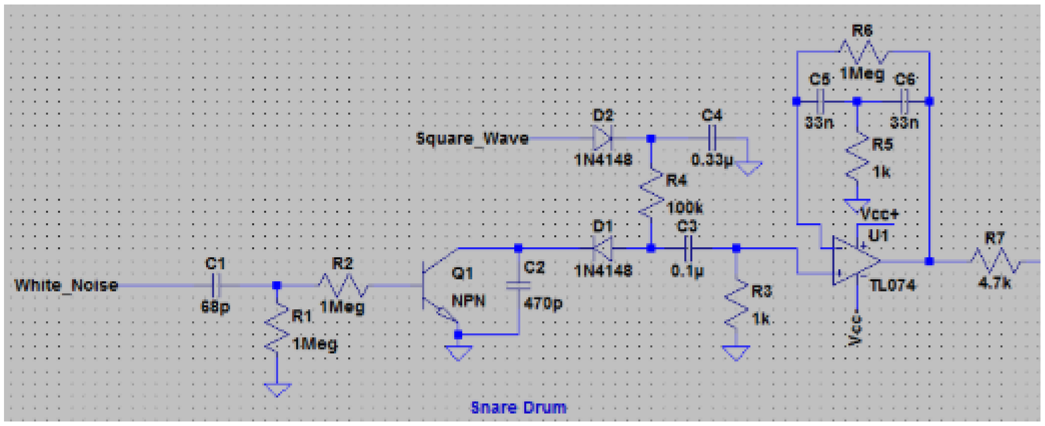


Figure 5 - Snare Drum Circuit Diagram

Physically, the sound of a snare drum is two sounds occurring simultaneously: the sound of the initial drum hit and the sound of the bottom drum head vibrating in resonance. These sounds are replicated in the above circuit in two separate stages. Initially, the input white noise pulse from the Arduino Uno is low-pass filtered by C1 and R1 (shown above in Fig. 5) before passing through a single NPN transistor, in a configuration that Roland's engineers employed in the TR-808, and referred to as a "Swing-Type Voltage Controlled Attenuator (VCA)," [8]. Figure 6 below shows how the Swing-Type VCA combines an envelope created by RC network (R4 and C4, shown above in Fig. 5), with the signal generated through filtering and attenuating the input white noise.

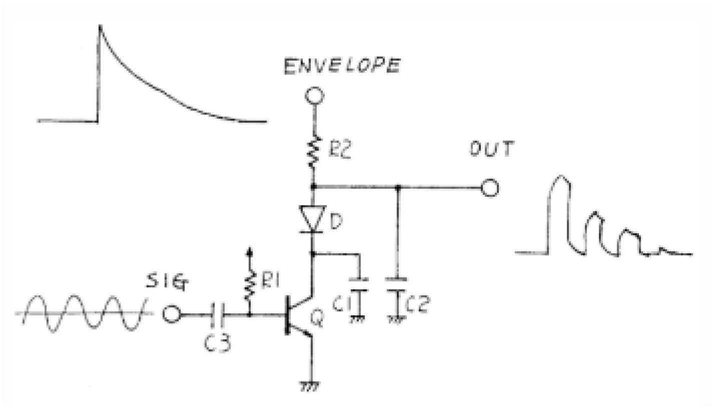


Figure 6 - Explanation of the operation of Roland's "Swing-Type VCA." This figure has been reproduced from the TR-808 Service Manual [9].

Additionally, a bridged t-network (R5, R6, C5, and C6 shown above in Fig. 5) is employed to created decaying sine waves, which are combined with the Swing-Type VCA signal by a TL074 (low-noise JFET operational amplifier). Further explanation of the bridged t-network follows below as this same circuit configuration creates the rim click, low tom, high tom, and bass drum sounds as well.

Rim Click

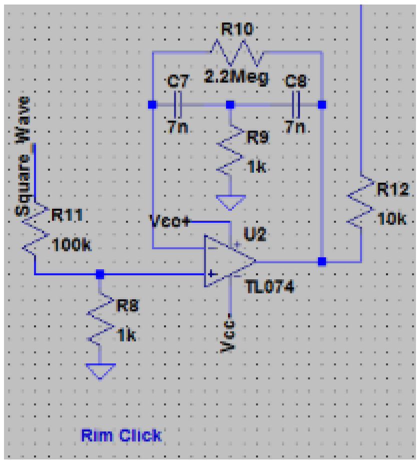


Figure 7 - Rim Click Circuit Diagram

As previously stated, a bridged t-network (R9, R10, C7, and C8 shown above in Fig. 7) creates the rim click sound in the E.D.M. Essentially, the bridged t-network employs a simple filter, which produces a decaying sine wave when excited into self-oscillation by a square wave. In the E.D.M., this square wave is provided by the Arduino Uno once a user presses the specific button designated for the rim click sound. Fig. 8, shown below, diagrams the function of a bridged t-network in greater visual detail, while also noting the equations used to determine the frequency of the output sine wave and the overall quality factor (Q) of the filter.

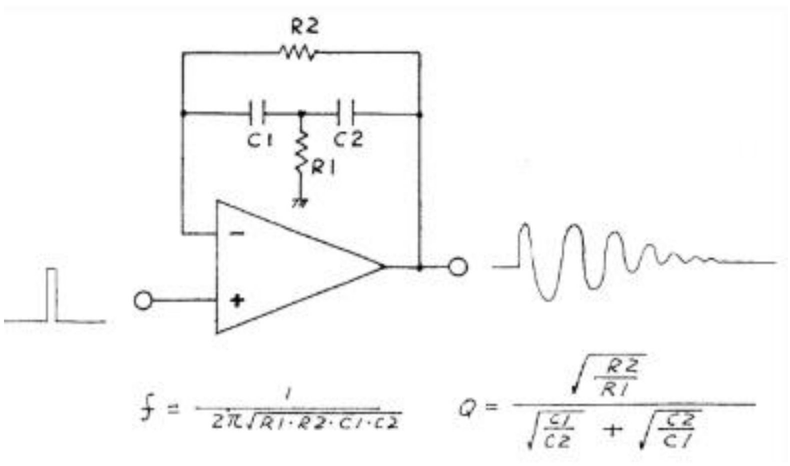


Figure 8 - Explanation of the operation of a bridged t-network, with relevant equations. This figure has been reproduced from the TR-808 Service Manual [9].

Low Tom, High Tom, and Bass Drum

The individual circuits that craft the low tom, high tom, and bass drum sounds are simply bridged t-networks with varying resistor and capacitor values. The differences in these values were determined by Delphtronics' Mickey Delp in the creation of his LDB-1 and then published online for free use by all. Diagrams for each individual circuit, as well as a final diagram demonstrating the connections between each individual circuit, are shown below.

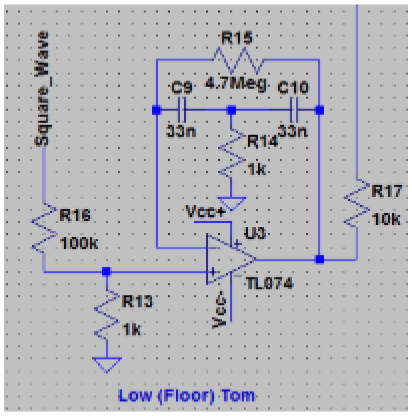


Figure 9 - Low Tom Circuit Diagram

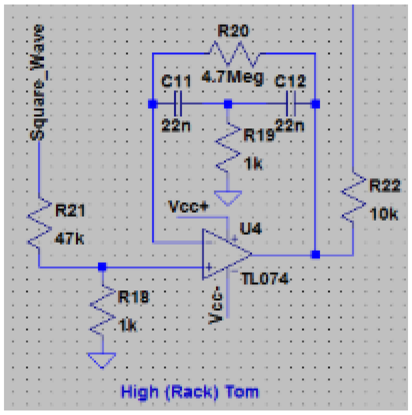


Figure 10 - High Tom Circuit Diagram

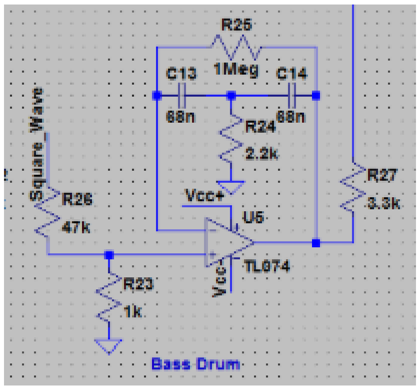


Figure 11 - Bass Drum Circuit

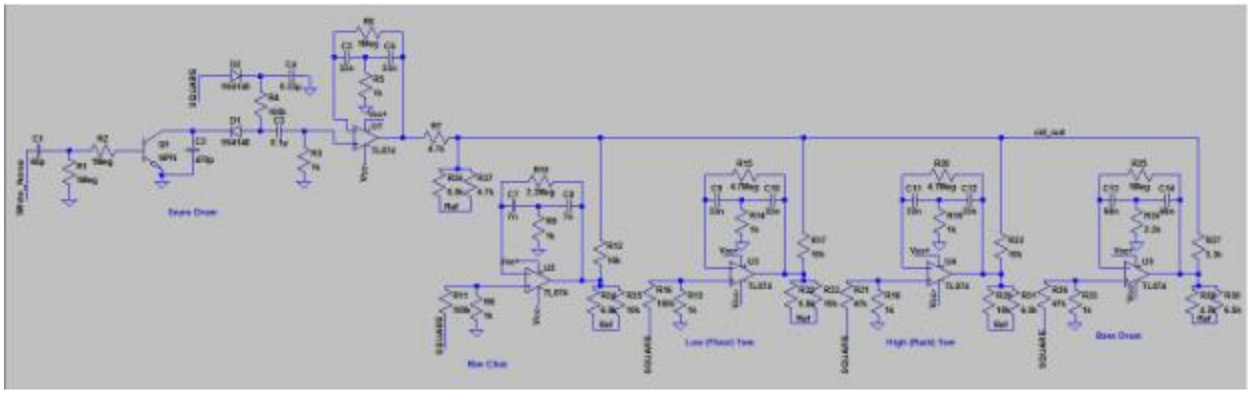


Figure 12 - Complete Hardware Circuit Diagram

Memory Configuration
Before saving any of the pre-defined drum patterns, or configuring a method to save the user-defined drum patterns, the software generated drum sounds had to be saved to the E.D.M's memory. The samples were originally recorded with small digital recorder and an actual drum set and professional drummer. Next, these samples were converted from an .MP3 format to a .WAV format and re-mixed from stereo-audio sampled at 44.1 kHz to mono audio sampled at 8,000 Hz using the freeware program Audacity. Finally, the samples were processed using the MATLAB program titled "convert_samples.m", provided in Appendix E. This program successfully converts each .WAV files

into an array of hexadecimal values. Additionally, “convert_samples.m” effectively resamples the audio file so that the output array is of a suitable size for saving to memory. Table XI, shown below, provides the final sample sizes in hexadecimal.

Table XI: Software-Generated Sample Sizes

Sample Name	**Sample Size (bytes)**
High Hat Cymbal(Foot)	0x16B
High Hat Cymbal(Open)	0x7EC
High Hat Cymbal(Closed)	0x201
Crash Cymbal	0x1F2
Ride Cymbal	0x1BD
Ride Cymbal Bell	0x156

For ease of access these samples were saved to an external serial EEPROM chip (24LC256) using the program titled “save_samples.c” included in Appendix E. A diagram showing the connections between the MSP432 and the serial EEPROM chip is included below (see Figure 13).

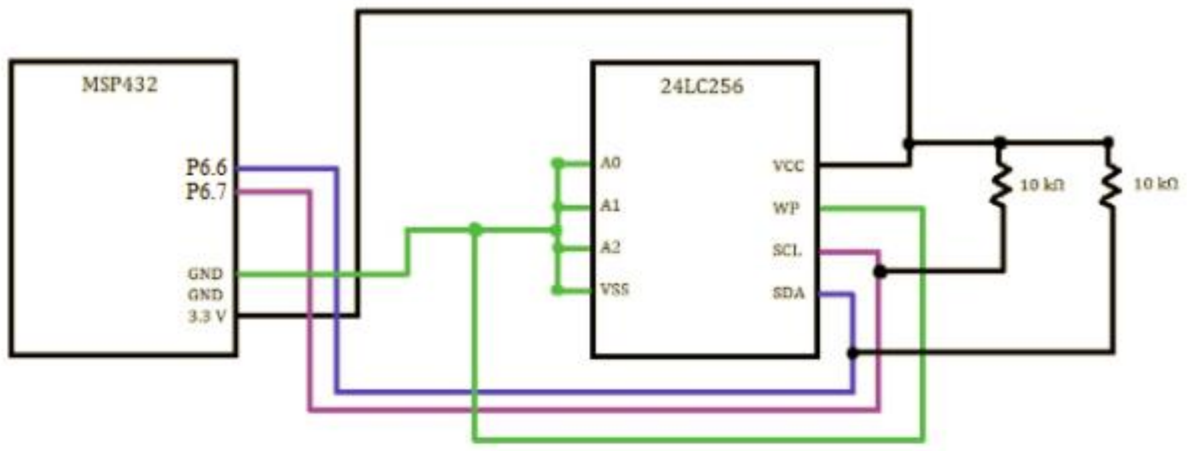


Figure 13 – The above diagrams shows all of the connections between the MSP432 and the serial EEPROM chip used to store the software-generated sounds and the pre-defined drum patterns.

The aforementioned process was also used to process and save 3 pre-defined drum patterns to memory. These drum patterns were downloaded from an open-source site [10]. While the initial customer requirements and specifications, provided above in Table I, specified that the E.D.M. must be capable of storing at least 20 pre-made drum tracks, due to time constraints only 3 patterns were saved as the process of converting and processing the patterns was too time intensive to save more patterns and still meet the project’s deadline. As before, for ease of access with an external button, these patterns were saved to the same external serial EEPROM chip as before. The names and final sizes of the pre-defined drum tracks are shown below in Table XII.

Table XII: Pre-defined Drum Pattern Sizes

Pattern Name	**Pattern Size (bytes)**
Aerosmith – “Walk This Way”	0x2BB
Black Sabbath – “Behind the Wall of Sleep”	0x24C
James Brown – “Funky Drummer”	0x19A

Initially, the user-defined patterns, including information regarding the tempo of each drum sample included in the user-defined pattern, were to be stored in the MSP432’s flash memory. However, as a reliable tempo control function was never successfully implemented, the E.D.M. is only capable of storing drum samples at one tempo layered over each other in the MSP432’s flash memory. While this short-coming greatly limits functionality, time constraints during the completion of the E.D.M. made this sacrifice necessary to complete all other features.

Output Circuitry

As stated in the customer requirements and specifications list, provided above in Table I, the E.D.M. must be capable of providing output via a MIDI jack and a 3.5mm jack. The necessary circuitry for these two output types were combined onto a single PCB, the schematic of which is shown below in Figure 13. The layout of this and all other PCBs included in the E.D.M. is included in Appendix D. Additionally, the individual circuits necessary for each of these output types are described in greater detail below.

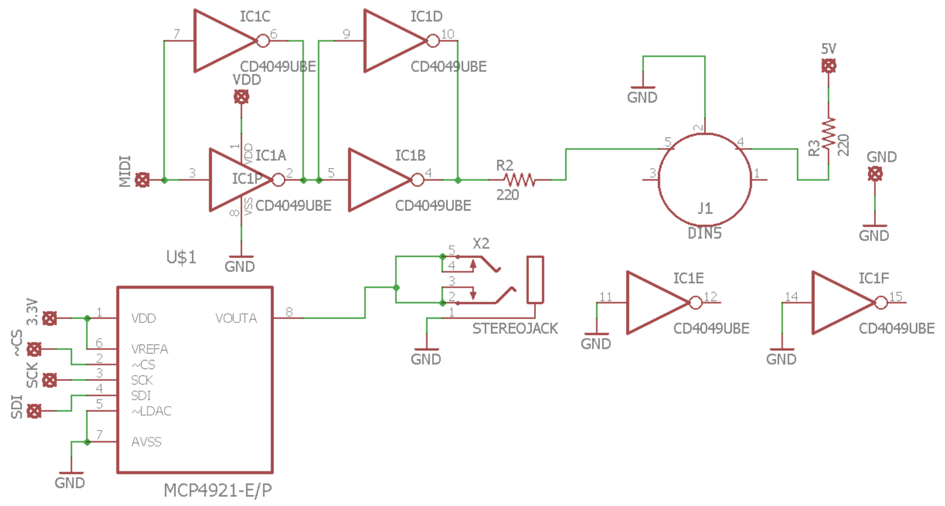


Figure 14 – Schematic of Output PCB, featuring circuitry for both MIDI output and 3.5mm output functionalities.

It should be noted that the above schematic features 4 inverters configured into 2 cascaded parallel pairs (IC1A, IC1B, IC1C, and IC1D). The parallel pairs were included simply because there were additional inverters in the CD4049 package that were not required for the MIDI circuitry and such an implementation simply increases the drive capability of the inverters, which does not affect anything in this circuit. Furthermore, the 2 parallel pairs are cascaded as a means of buffering between the MSP432's output pin and the input of the DIN jack.

MIDI Output

All of the circuitry required for a MIDI output was developed using the MIDI DIN Electrical Specifications so that the E.D.M. will properly interface with any other consumer devices that support MIDI [11]. The circuitry necessary for a MIDI output is shown below in Figure 14. However, it should be noted that IC1E and IC1F (the 2 uncommitted inverters shown in the PCB schematic in Figure 13) are not reflected anywhere in Figure 14. This is because these 2 inverters are not necessary for the MIDI output circuit and simply remain on the PCB with their inputs grounded and their outputs unconnected.

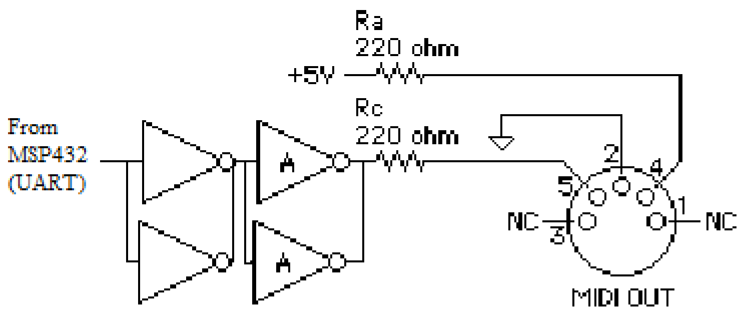


Figure 15 – MIDI Output Circuit. The above image was adapted from the MIDI DIN Electrical Specifications to more closely reflect the circuit implemented in the E.D.M [11].

3.5mm Output

The 3.5mm output is provided through a 3.5mm mono TRS (tip-ring-sleeve) jack. This is nearly identical to the jack that most consumers plug headphones into in a laptop or smartphone and functions in the exact same manner. That is, the output signal must be delivered to the TRS jack as an analog voltage [12]. Since the MSP432 is a digital system, and until this output stage all of the drum patterns have been represented as digital signals, a digital-to-analog converter (DAC) is needed between the output of the MSP432 and the input of the TRS jack to ensure proper function. The circuitry, and programs, used to configure a DAC chip (MCP4921) were borrowed from a similar assignment completed in CPE 329. The MCP4921 is a 12-bit DAC, which means that some resolution of the hardware generated sounds will be lost as the hardware-generated sounds were sampled with a 14-bit ADC. Yet, this was deemed an acceptable loss as the software-generated samples and pre-defined patterns have been configured with an 8-bit resolution, making a 14-bit DAC unnecessary for the majority of possible outputs. The necessary circuit is shown below in Figure 15 and all of the necessary code is provided in Appendix E.

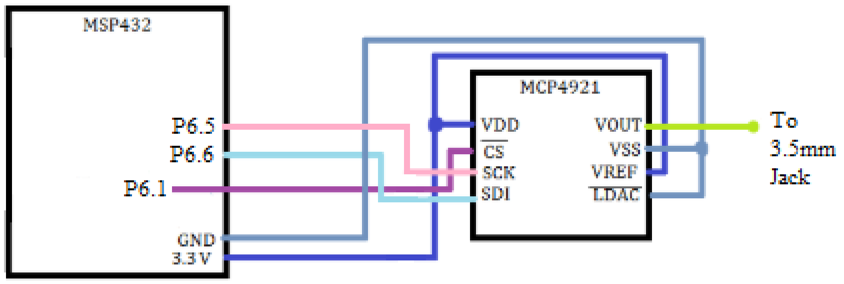


Figure 16 - DAC Circuit Diagram

Power
Initially, the E.D.M. was intended to utilize either AC wall power or battery power, yet, due to time related constraints the finally design only utilizes battery power.
Throughout the totality of the E.D.M. several DC voltages are required. For instance, the Arduino Uno needs a 5V supply for power, while the MSP432 needs a 3.3V source. The PCB that provides the hardware-generated sounds requires a 10V input, and a separate 1.5V source for offsetting the resultant signals before ADC in the MSP432. Finally, the output circuitry requires both a 3.3V input for the 3.5mm jack output circuitry and a 5V input for the MIDI output circuitry.
Thus, for simplicity, a 12V DC source (8 AA batteries) was introduced and divided into the proper voltages using the simple resistive network shown below in Figure 16. Such a configuration also allows for a straightforward common ground connection between the Arduino Uno, the MSP432, and all of the PCBs and additional circuitry. While this solution certainly invites future errors, especially as the batteries begin to lose charge, this solution was chosen as the most schedule effective of all possible options. A separate 1.5 V (1 AA battery) source was included to provide the offset for the hardware-generated sounds PCB.

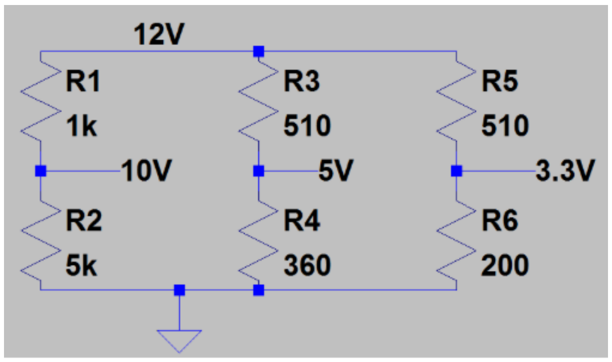


Figure 17 – Resistive network used to divide 12 V DC into necessary voltages.

IV: Construction

Display and User Interface (UI)

As the E.D.M. is intended to be approachable and user-friendly, regardless of a potential user's musical or technical background, display and user interface (UI) were implemented in a minimalist fashion. Thus, the display consists of a single LCD capable of displaying 2 rows of 16 characters each, and the UI consists of three separate panels of buttons and an on-off switch. To further increase usability, each button is clearly labeled with its function. A photograph of the E.D.M's UI is shown below in Figure 17. It should be noted that the on-off switch is located on the left lateral side of the E.D.M. and, as such, is not included in the photograph in Figure 17.

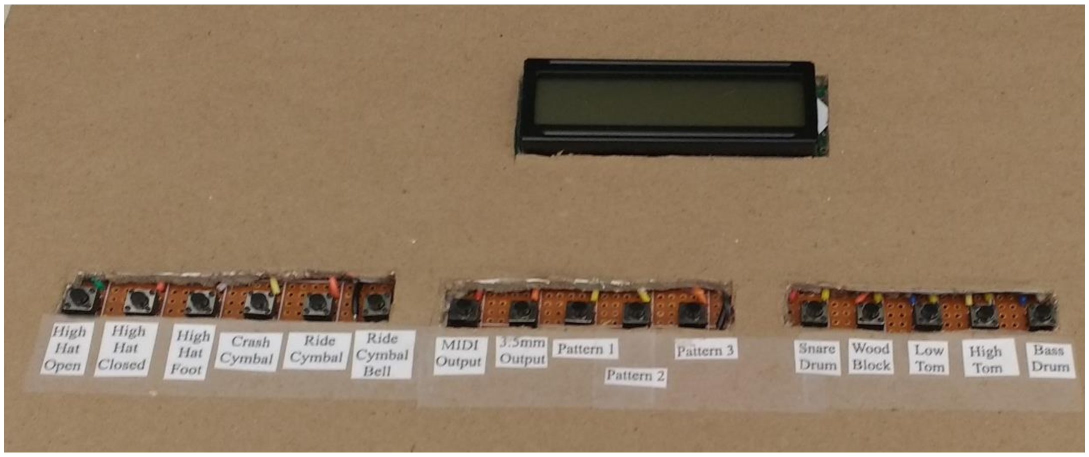


Figure 18 – This photograph shows the top of the E.D.M, which includes the LCD and the totality of the functional buttons.

Chassis Design

Initial specifications provided above in Table I suggest that the E.D.M. derive its highly durable character from a metal chassis. As with several other aspects of this project, the approaching deadline did not afford the time necessary to design and build such a complex chassis. Instead, a cardboard box, purchased from a craft store and originally intended as decorative storage, served as the chassis for the final design.

In many ways, choosing to use a cardboard box served to simplify the installation of the completed electronics into the chassis as it allowed for simple chassis customization. Simply put, I was able to cut all of the necessary holes in the box using a craft blade. Furthermore, the cardboard box allowed me to affix the circuit boards to the box using hot glue and a variety of spacers. Originally, nylon spacers were to be employed to hold up the two custom PCBs. Yet, due to the unusual dimensions of the cardboard box, and time constraints, I could not procure spacers of an adequate length, so small balsam wood dowels and hot glue were used instead.

The following figures (18 – 21) show all of the relevant features of the final chassis design and implementation.

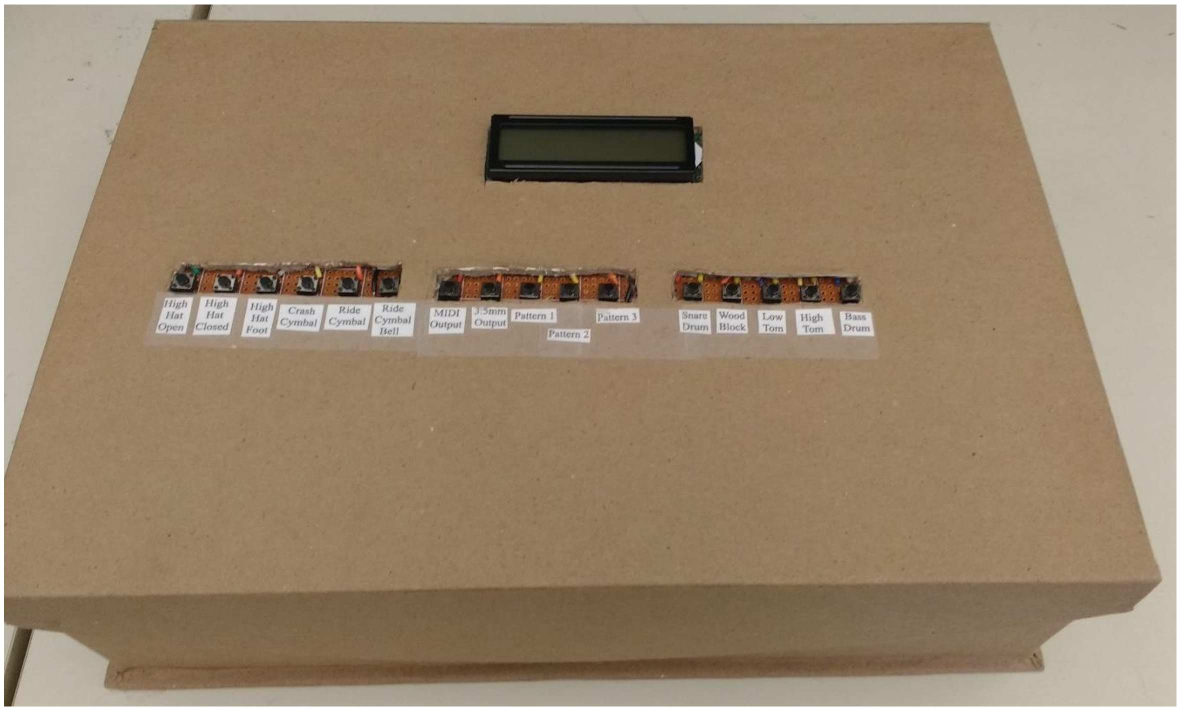


Figure 19 – This photograph again shows the top of the final chassis, with the LCD and labeled button panels.

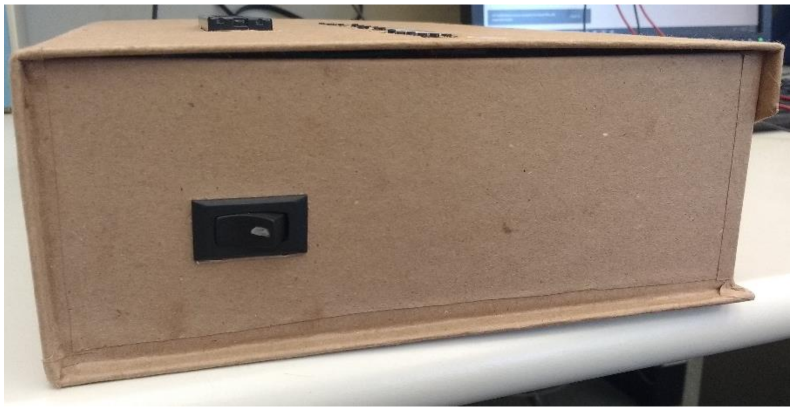


Figure 20 – The above photograph shows the E.D.M's on-off switch, with the silver mark denoting the switches ON position.

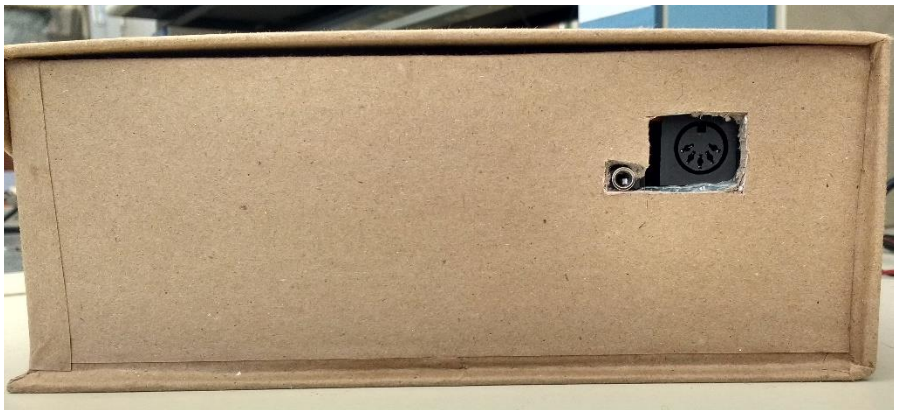


Figure 21 – This photograph shows the E.D.M's two output ports. The small silver circle to the left is the 3.5 mm output, with the larger black circle to the right is the MIDI output.

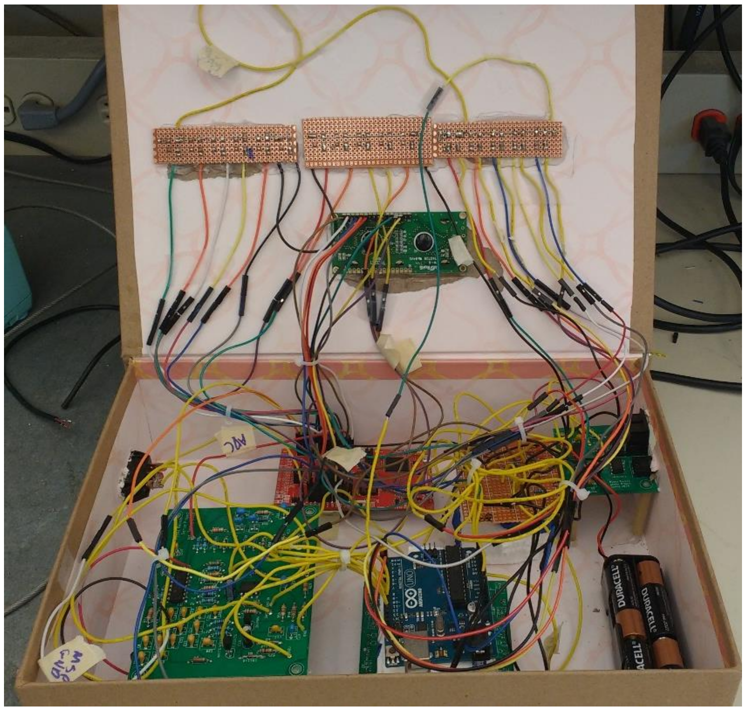


Figure 22 – The above photograph shows the totality of the electronics of the E.D.M, affixed inside of the chassis.

PCB Design

The PCB for the hardware-generated sounds was completed first and required a revision and re-fabrication after testing the first design. This is because the hardware-generated sounds PCB features a split ground plane, which was implemented incorrectly in the first iteration.

While both iterations of the hardware-generated sounds PCB utilized split ground planes, the two boards did so for different reasons. In the first iteration, a split ground plane was (incorrectly) implemented to provide an isolated ground for the circuit that generates the rim click sounds. This was because that circuit was found to be quite noisy, and the noise it introduced to the system caused unwanted errors in the other circuits. The layout of the first iteration is shown below in Figure 22.

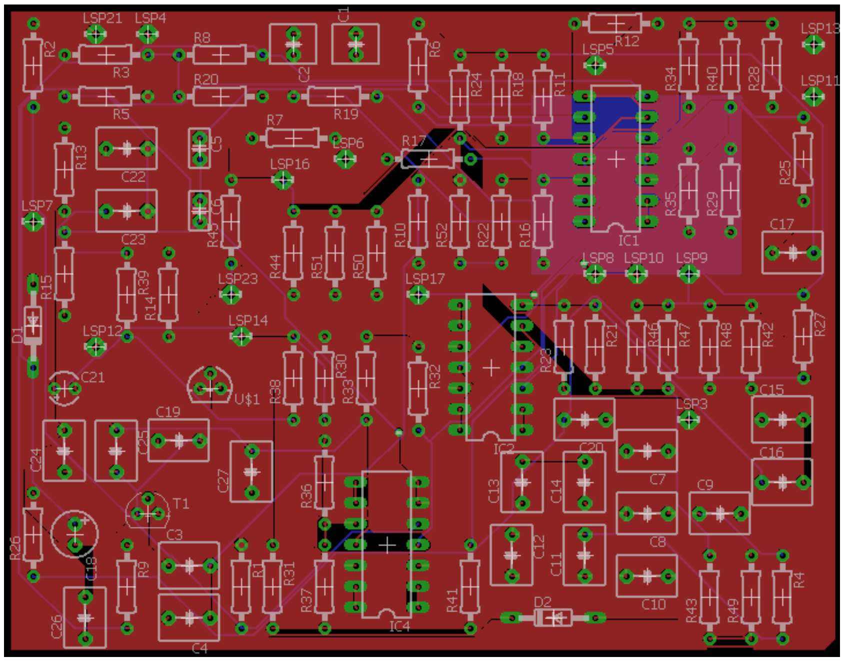


Figure 23 – The above image shows the first design iteration of the hardware-generated sounds PCB. The split ground plane can be seen in the upper right corner as a light blue box.

However, after fabricating and testing this first PCB a different design was employed to eliminate noise in the second iteration. The second iteration eliminated noise in the rim click circuit by isolating the rim click circuit to its own op-amp chip. Effectively, this removed the need for a split ground plane between the rim click circuit and the rest of the circuits as the noise was eliminated by the isolation. Yet, a split ground plane was implemented in this final version as a means of isolating analog and digital grounds. Thus, the microcontroller connections have a separate ground plane from the circuits to eliminate any potential ground noise interference between the analog and digital parts of the PCB. The final PCB schematic and layout are included below in Appendix E.

After the completion of the hardware-generated sounds PCB, a PCB for the output circuitry was created. This PCB only required one version as it is a far simpler design and does not require a split ground plane. The layout and schematic of this PCB are also included below in Appendix E.

V: Testing

Arduino Buttons

Initially, Arduino programs that generated square waves, or white noise, dependent on external button presses were constructed. Yet, these programs had to be modified once the Arduino was integrated with the hardware-generated circuits PCB and the MSP432. Modifications had to be made because the MSP432's ADC was configured to collect and convert data for 300 ms, as this length of time is slightly longer than the longest hardware-generated circuit pulse. While the MSP432 was configured to collect data for 300ms, the previous Arduino programs continued to excite the hardware-generated circuits PCB until a second button press was detected. This led to a mis-match between the Arduino and the MSP432 where, for even numbers of button presses, the circuit would be excited without the MSP432's ADC being turned on. Thus, the Arduino programs were edited to generate 300 ms of either square waves or white noise pulses for each button press. The final version of the complete Arduino program is included in Appendix E.

Furthermore, Figure 23, shown below, shows how the individual button connections on the Arduino are configured. It is important to note that the internal pull-up resistors of both the Arduino Uno and the MSP42 have been enable in software so that both microcontrollers could detect the same button press.

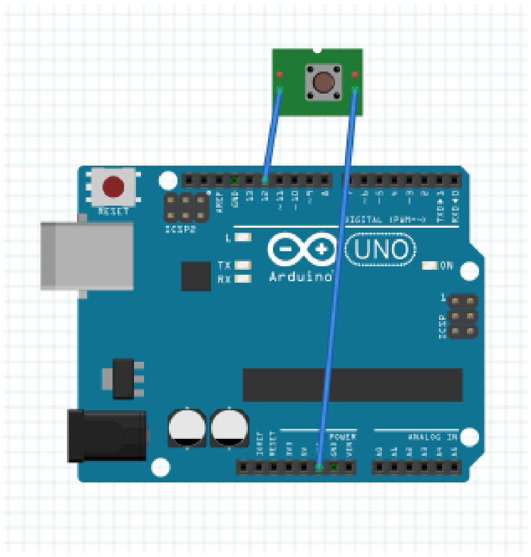


Figure 24 - The above diagram shows an example of how each external button is connected to the Arduino Uno.

Additionally, as pull-up resistors are being used, the microcontrollers (MSP432 and Arduino Uno) are essentially "waiting" for the button pins to be pulled high (to the supply voltage), indicating that the button has been pressed. The following diagram, shown in Figure 24, shows the microcontroller's pin with the button connection and internal pull-up resistor. It is important to note that this diagram holds true for the button connections to the Arduino Uno and the MSP432.

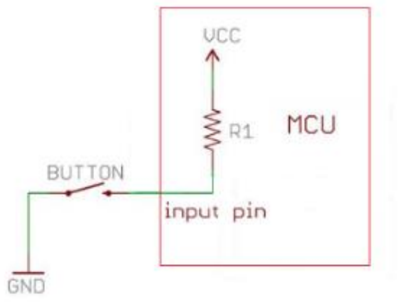


Figure 25 - The above diagram shows the connection between and external button and a microcontroller's pin with an internal pull-up resistor. This diagram has been modified from a diagram featured in Sparkfun's Pull-Up Resistor Tutorial [13].

Hardware Generated Sounds

Before any physical circuit testing occurs, the circuits required for the 5 hardware generated sounds undergo thorough testing in LT Spice simulation. This process vets the initial design of any obvious errors and provides a reference output with which to compare the output of the physical testing stage. The output of the LT Spice simulation of each individual circuit, when provided a 5 V_{P-P} square wave input at 4 Hz, is provided below.

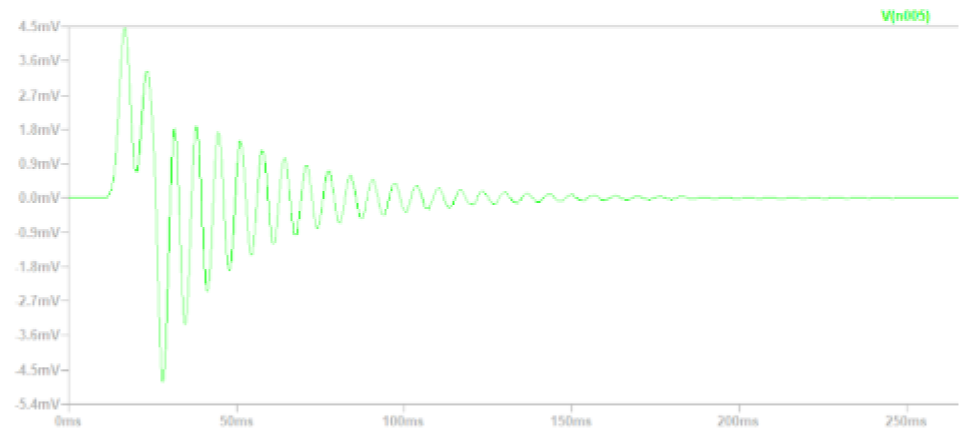


Figure 26 - The above plot shows the output of the LT Spice simulation of the snare drum circuit.

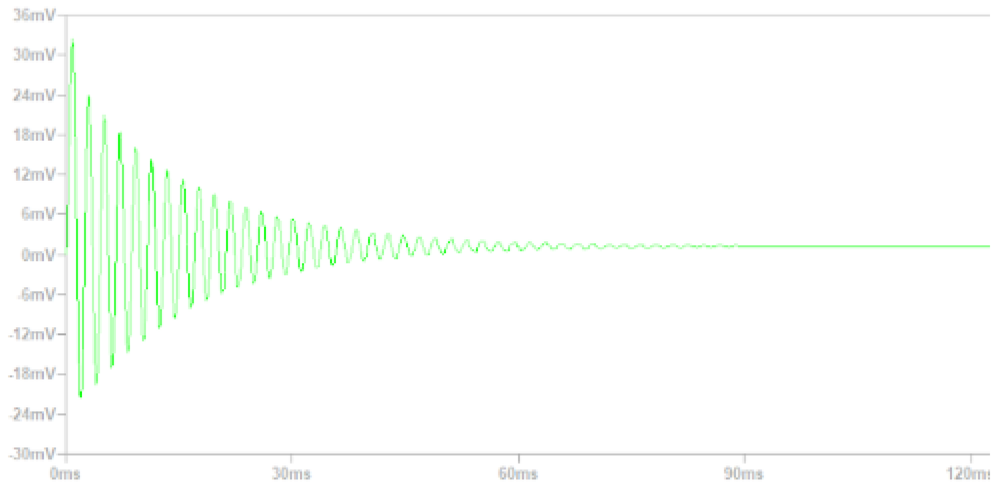


Figure 27 - The above plot shows the output of the LT Spice simulation of the rim click circuit.

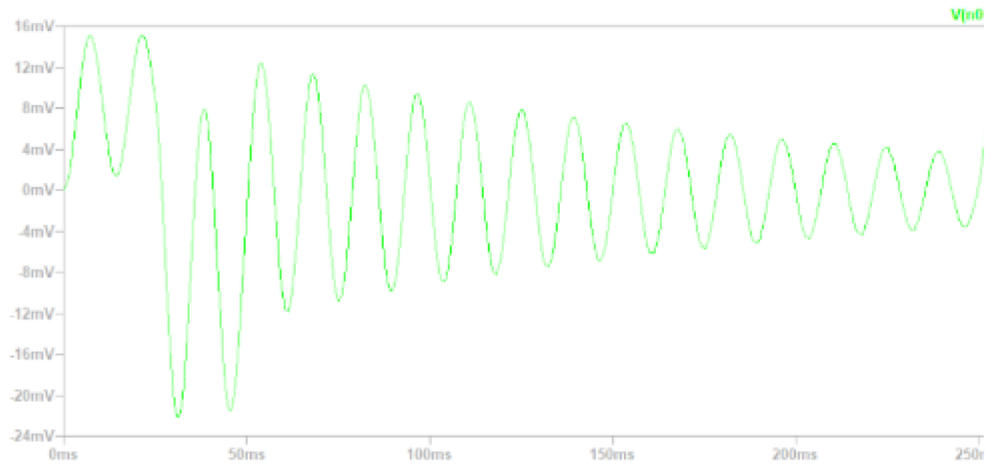


Figure 28 - The above plot shows the output of the LT Spice simulation of the low tom circuit.

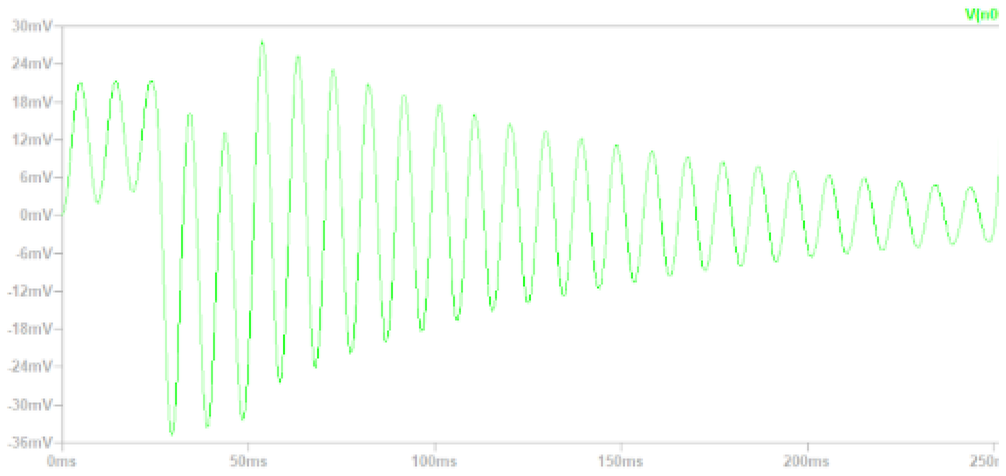


Figure 29 - The above plot shows the output of the LT Spice simulation of the high tom circuit.

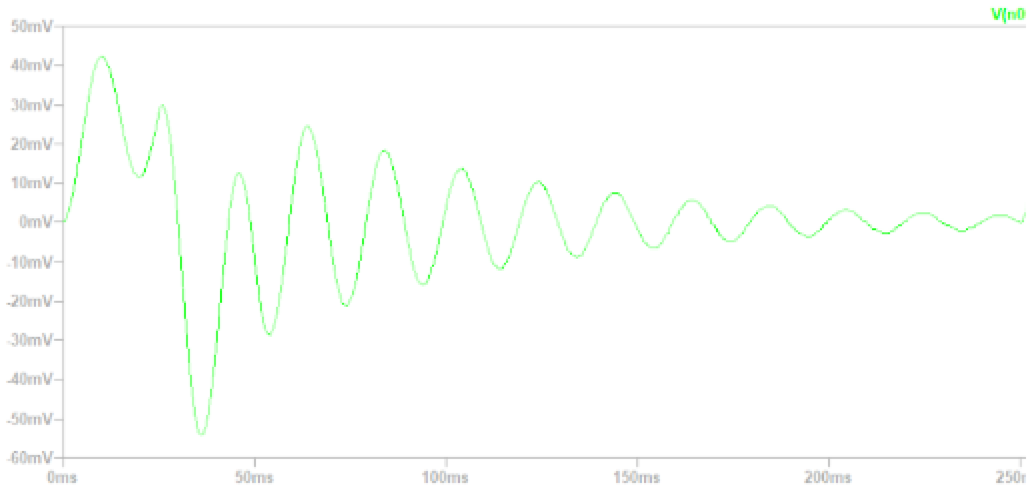


Figure 30 - The above plot shows the output of the LT Spice simulation of the bass drum circuit.

Following the completion of the LT Spice simulations, the previously-constructed square wave and white noise outputs of the Arduino Uno were used to test bread board models of the physical circuits. The results of these tests, as shown on an oscilloscope display, are provided below.

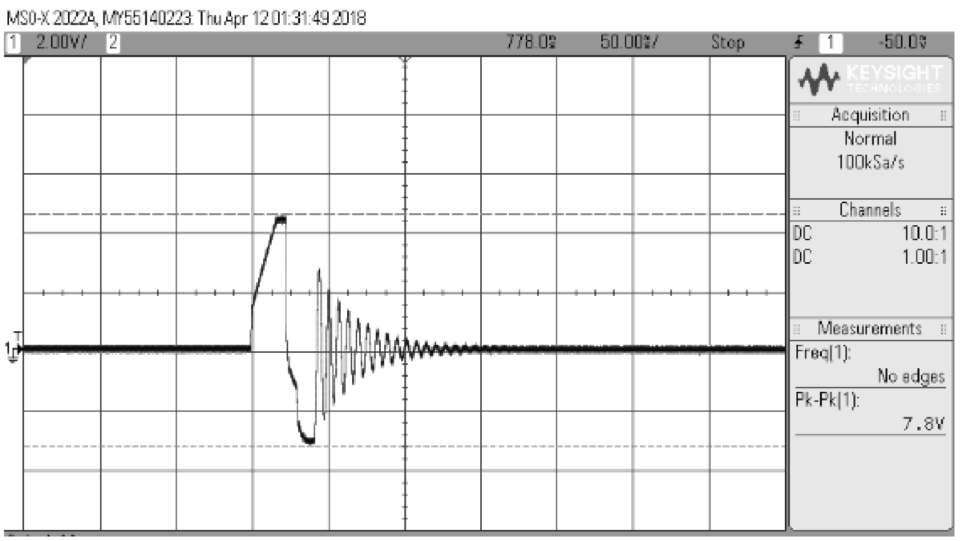


Figure 31 - The above oscilloscope capture shows the output of the bread-boarded snare drum circuit, when excited by a square wave and white noise pulse provided by the Arduino Uno.

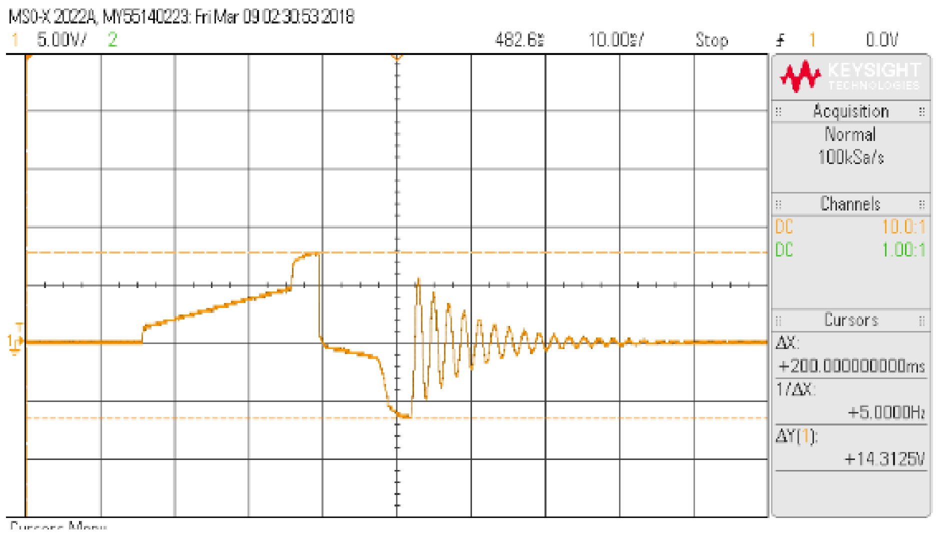


Figure 32 - The above oscilloscope capture shows the output of the bread-boarded rim click circuit, when excited by a square wave provided by the Arduino Uno.

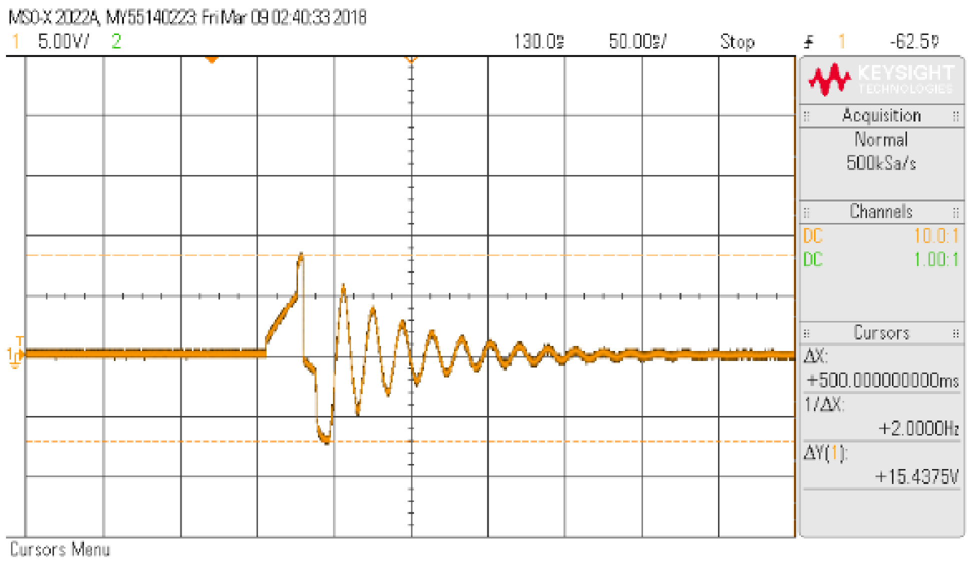


Figure 33 - The above oscilloscope capture shows the output of the bread-boarded low tom circuit, when excited by a square wave provided by the Arduino Uno.

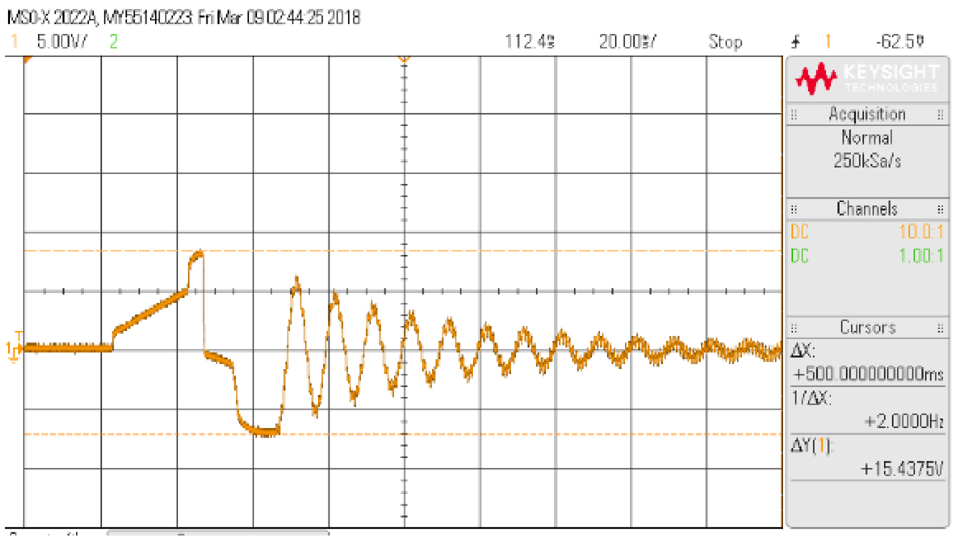
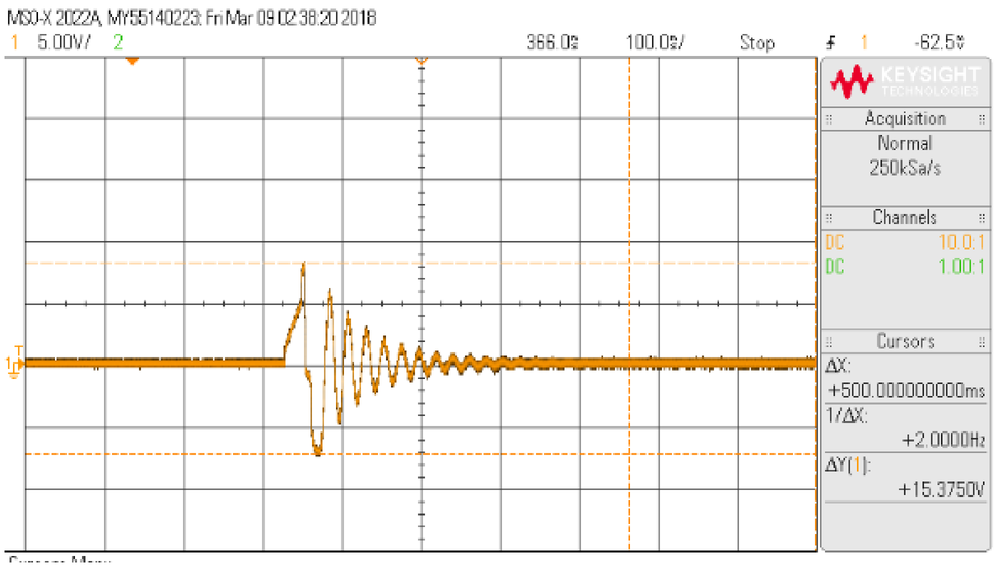


Figure 34 - The above oscilloscope capture shows the output of the bread-boarded high tom circuit, when excited by a square wave provided by the Arduino Uno.


MSO-X 2022A, MY55140223: Fri Mar 09 02:38:20 2018

1 5.00V/ 2 366.0s 100.0s/ Stop 1 -62.5

Acquisition
Normal
250kSa/s

Channels
DC 10.0:1
DC 1.00:1

Cursors
ΔX:
+500.000000000ms
1/ΔX:
+2.0000Hz
ΔY(1):
+15.3750V

Figure 35 - The above oscilloscope capture shows the output of the bread-boarded bass drum circuit, when excited by a square wave provided by the Arduino Uno.

Thus, after examining the results of the aforementioned testing procedures, it was concluded that the bread-boarded physical iteration of the hardware circuits functioned as predicted by simulation.

MSP432 Analog-to-Digital Conversion (ADC)

Once the hardware generated sounds are generated by their respective circuits, the resulting analog output signals must then be converted into a digital form and stored in the MSP432 for further use and manipulation. Thus, following the completion of the circuits necessary to generate the hardware generated sounds, the MSP432's ADC must be configured and tested to ensure sufficient operation.

Testing and proper configuration of the MSP43's ADC began with the development of a program in Code Composer Studio (CCS), a development environment that supports the MSP432 and other of Texas Instruments' microcontrollers, which detects an analog voltage on one of the MSP432's dedicated ADC pins and reports the resulting digital value to the terminal via UART. Parts of this testing program were developed several months ago, as part of a programmable logic and microprocessor-based systems course at Cal Poly (CPE 329). As such, acknowledgement is due to my lab partner in that course, Andrea Levy, who contributed significantly to this program, and all other programs constructed over the duration of that course. The complete testing program, titled "ADC_TEST.c", is included in Appendix E. The following image provides a diagram of the MSP432's pins, including those designated as analog inputs for ADC. Additionally, the following image shows which pins were used for this testing.

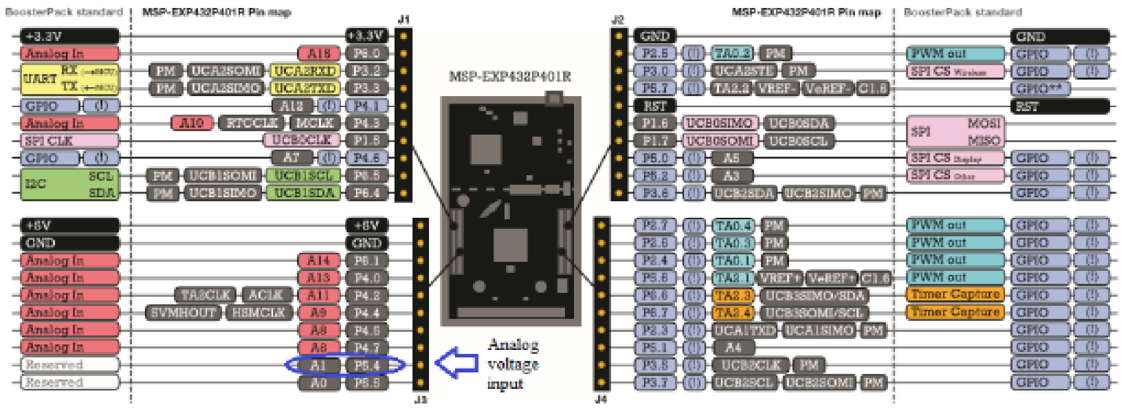


Figure 36 - The above image shows the pinout of the MSP432, with annotations showing which pins were used for the ADC testing and configuration [14].

After confirming that the testing program implemented the ADC functionality as intended, the accuracy of the ADC was tested. Accuracy testing began by plotting the digital values, in millivolts, reported by the MSP432 against the sample number and the viewing the resultant graph in Excel. Ideally, such a process should result in a graph that exactly mimics the input waveform which, in the case of this test, was a 2 V_{P-P} sine wave at 4 Hz. The ADC graph before any calibration is shown below in Figure 37.

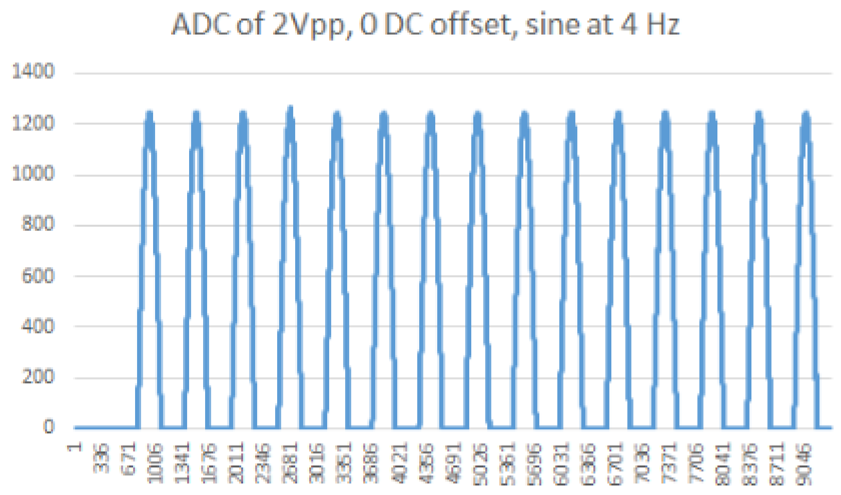


Figure 37 - The above graph shows the calculated digital voltages in millivolts versus sample number when a 2 V_{P-P} sine wave at 4 Hz is input to the MSP432's ADC.

As the above graph does not accurately match the analog input, since the peaks should occur at 1000 mV instead of slightly above 1200 mV, calibration is required to increase the ADC's accuracy. The necessary scale factor is determined by applying a series of known voltages to P5.4 and recording the digital voltage returned to the terminal. The known and calculated values are then graphed against each other and the equation-of-best-fit of the resulting graph is the scale factor needed to correct the ADC. For this particular ADC, applying the determined scale factor changed one line of code in "ADC_TEST.c." The line `millivolt = (int) var` was changed to `millivolt = (int)((var + 1.46) / 1.2361)` to calibrate the ADC. The calibrated ADC output, under the same input conditions as described in Figure 37 is shown below in Figure 38**.**

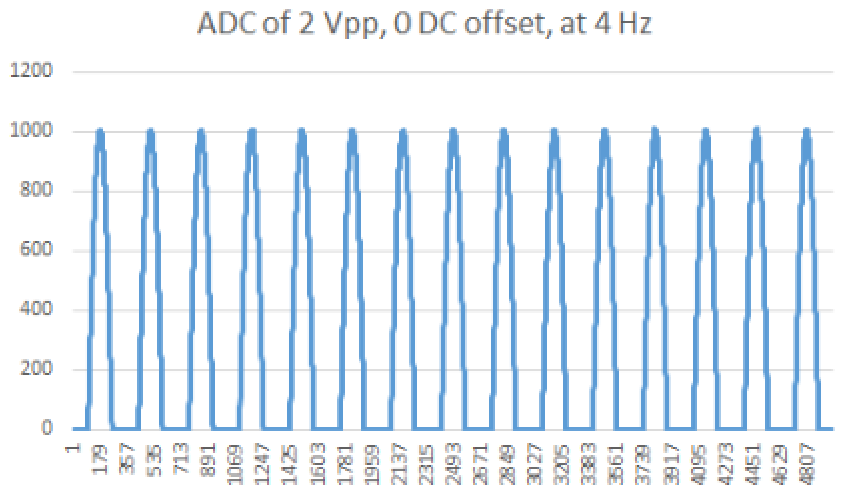


Figure 38 - The above graph shows the calculated digital voltages versus sample number after successful ADC calibration.

To prevent pin malfunction, voltages greater than 3.3 V should never be applied to any of the MSP432's pins while a 3.3 V power source is powering the microcontroller. As such, the input analog values need to range from 0 V, as the MSP432 does not easily convert negative voltages, to 3.3 V. Figures 31 - 35 show that the output waveforms from the circuits necessary for the hardware generated sounds do not meet this standard. Thus, voltage division between the output of the circuits and the input of the MSP432's ADC is necessary to prevent pin failure. Through extensive physical testing the resistor values used in each voltage divider, shown in Figure 38 as part pf the completed schematic for the circuits necessary for the hardware generated sounds, were determined. Once completed, each drum circuit and its corresponding voltage divider were again tested to ensure their output waveforms conformed to the necessary ADC input constraints. The results of these tests are shown below in Figures 41 - 45**.**

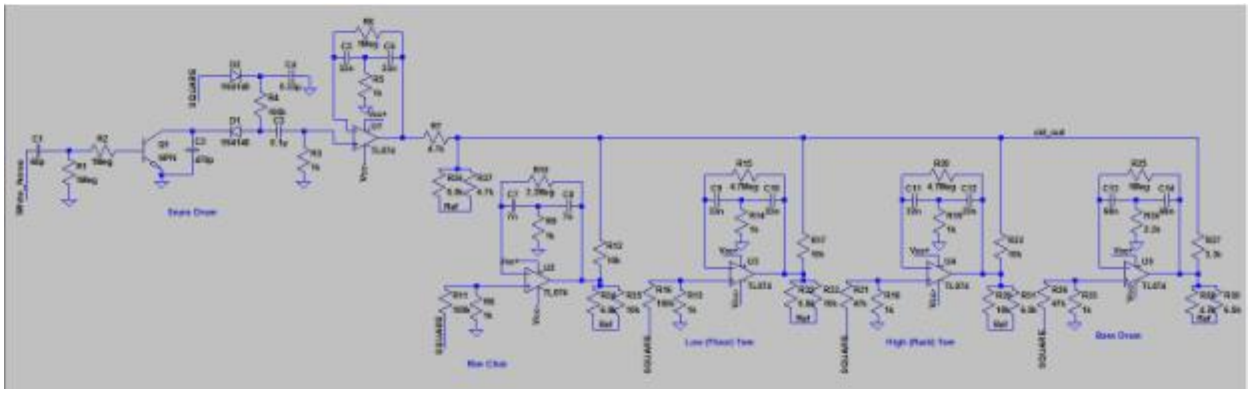


Figure 39 - The above schematic shows the completed circuits, including voltage dividers, necessary for generated all of the hardware dependent sounds of the E.D.M.

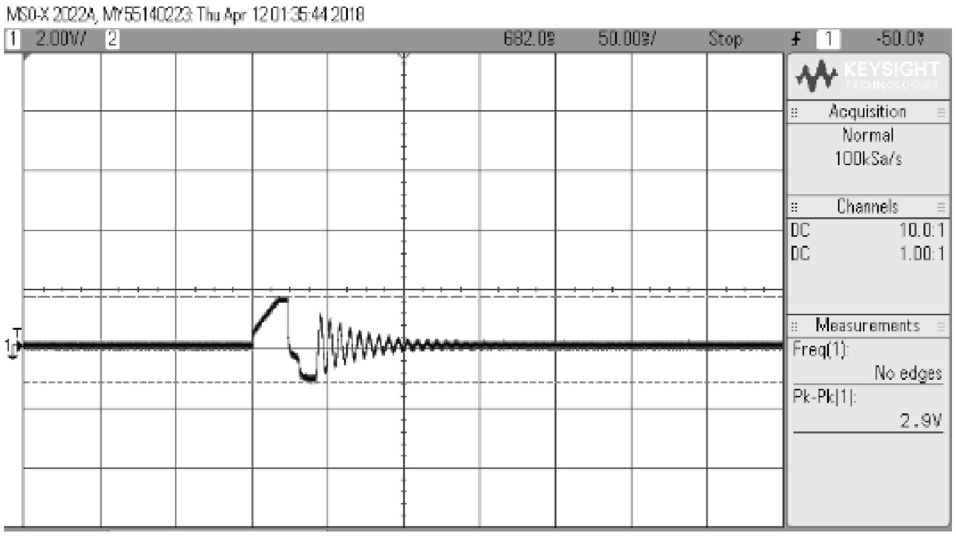


Figure 40 - The above oscilloscope capture shows the output of the completed snare drum circuit and voltage divider, when excited by a square wave and white noise pulse provided by the Arduino Uno.

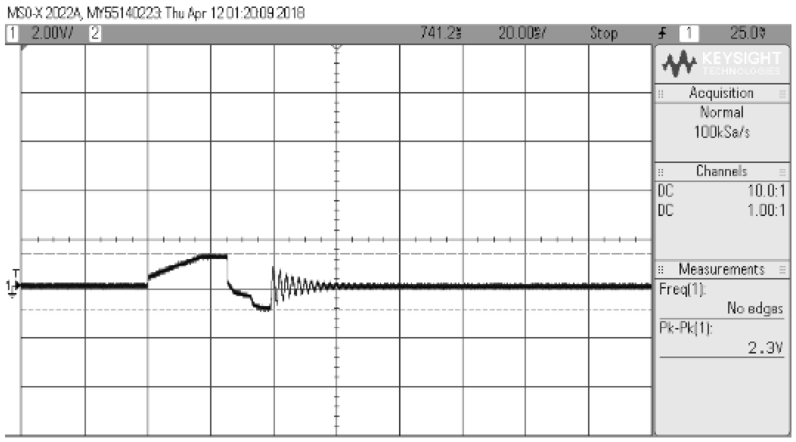


Figure 41 - The above oscilloscope capture shows the output of the completed rim click circuit and voltage divider, when excited by a square wave provided by the Arduino Uno.

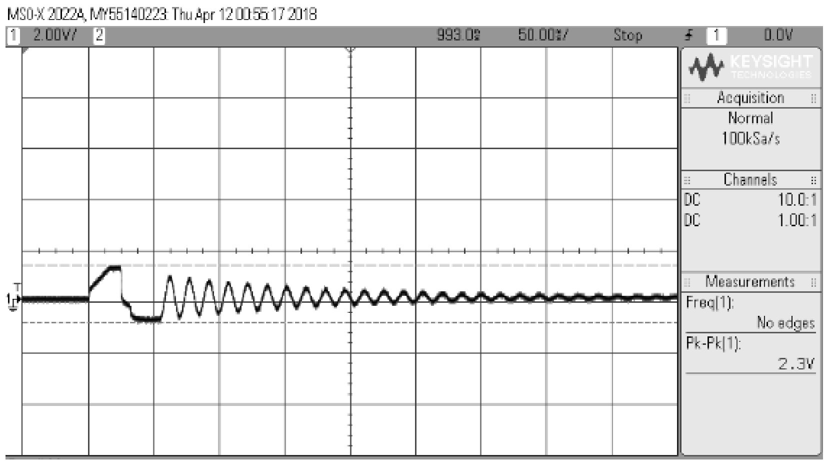


Figure 42 - The above oscilloscope capture shows the output of the completed low tom circuit and voltage divider, when excited by a square wave provided by the Arduino Uno.

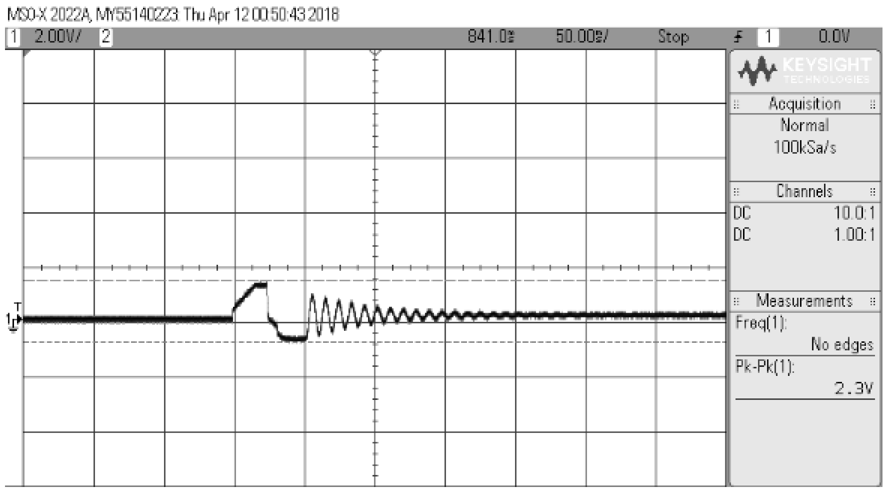


Figure 43 - The above oscilloscope capture shows the output of the completed high tom circuit and voltage divider, when excited by a square wave provided by the Arduino Uno.

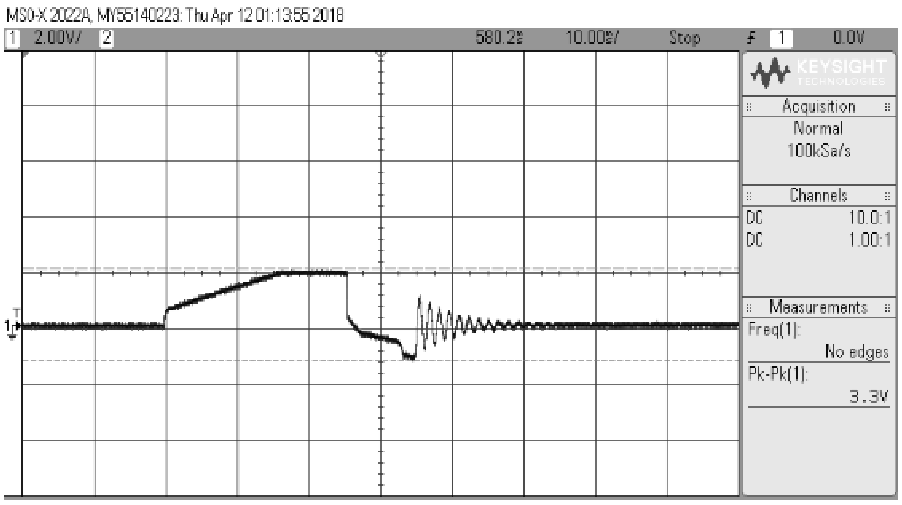


Figure 44 - The above oscilloscope capture shows the output of the completed bass drum circuit and voltage divider, when excited by a square wave provided by the Arduino Uno.

Additionally, all of the voltage division circuits will provide 1.65 V DC offset, so as to maximize the dynamic range of the input to the MSP432's ADC.

Memory
Once all of the individual drum samples (software-generated) and pre-defined patterns were written to the serial EEPROM chip, their existence in memory was validated using a program called "EEPROM_Test.c" (see Appendix E). ""EEPROM_Test.c" was used to check 25 random addresses in the address range of each drum sound or pre-defined pattern. The program waited until an external button was pressed and then returned the values stored at each of those 25 locations and these values were checked against the tables of values that were programmed at specific address for each drum sound or track.

Output Circuits
The 3.5mm output circuitry consists primarily of a digital-to-analog converter (DAC) connected between the MSP432 and the TRS jack. A program that uses the MSP432 to drive this specific DAC chip was developed during CPE 329. This program was used to verify proper DAC functionality and is included in Appendix E (see "DAC_Test.c"). Finally, upon completion of the E.D.M, 3.5mm output was verified by connecting the E.D.M. to a speaker and observing audio output.
Verification of the MIDI output circuit was facilitated by simply connecting the E.D.M. to a synthesizer that accepts MIDI input. This was because the MIDI circuitry is simplistic and only requires a UART signal from the MSP432. As UART signals were used to test many other components of the project thus far (including the ADC and 3.5mm outputs) UART transmission functionality was not tested independently to ensure the correct operation of the MIDI output.

LCD
A program that facilitates writing characters on a 2X16 LCD was developed during CPE 329 and then modified for use in this project. The program from CPE 329 was used as a means of testing the LCD, and ensuring that all connections between the MSP432 and the LCD are correct. This program is titled "LCD_Test.c" and is included below in Appendix E.

VI: Conclusions and Recommendations

In conclusion, the E.D.M. was an intensive and enjoyable project that successfully satisfied some of its initial requirements upon completion. The principal recommendation for future development is to develop a method of tempo control and improve the storage of user-defined patterns in the MSP432's flash memory. Additionally, if more time had been allowed for the completion of the E.D.M. different methods of storing the software-generated drum sounds and the pre-defined patterns would have been investigated and potentially implemented. This is because the EEPROM chip used significantly limits the size of the software-generated drum sounds, which reduces their quality. Furthermore, the method used to process the software-generated drum sounds and pre-defined drum patterns was unusually long and cumbersome, which prevented the addition of more than 3 pre-defined drum patterns. In the future, a new method of processing and storage should be implemented.

VII: References

[1] M. L, "How Much Does a Drum Set Cost? A Beginner's Guide," *takelessons.com*, para. 3, April 23, 2014. [Online]. Available: https://takelessons.com/blog/how-much-does-a-drum-set-cost. [Accessed: Nov. 4, 2017].

[2] R. B, "Ghost In the Machine: The Most Important Drum Machines in Music History," *complex.com*, para. 1, May 26, 2014. [Online]. Available: http://www.complex.com/music/2014/05/most-important-drum-machines/. [Accessed: Nov. 4, 2017].

[3] "Roland TR-808," Nov. 1, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Roland_TR-808. [Accessed: Nov. 5, 2017].

[4] Z. Hasnain, "How the Roland TR-808 Revolutionized Music," *theverge.com* April 3, 2017. [Online]. Available: https://www.theverge.com/2017/4/3/15162488/roland-tr-808-music-drum-machine-revolutionized-music/. [Accessed: Nov. 4, 2017].

[5] S. Wilson, "Buying a Drum Machine: What to consider and where to get the best deal," *factmag.com*, September 9, 2017. [Online]. Available: http://www.factmag.com/2017/09/09/best-drum-machines-advice-deals/. [Accessed: September 20, 2017].

[6] R. Ford and C. Coulston, *Design for Electrical and Computer Engineers*, McGraw-Hill, 2007.

[7] T. Page, "The 9090 project: the complete TR-909 Clone," *9090project.co.uk*, June 23, 2012. [Online]. Available: http://www.9090project.co.uk/. [Accessed: Nov. 9, 2017].

[8]Delp, M. (2012). *Anatomy of a Drum Machine*. [online] Anatomy of a Drum Machine. Available at: http://mickeydelp.com/blog/anatomy-of-a-drum-machine? [Accessed 6 Feb. 2018].

[9]TR-808 Service Notes. (1981). 1st ed. [PDF] Roland, pp.1-16. Available at: http://manuals.fdiskc.com/tree/Roland/Roland%20TR-808%20Service%20Manual.pdf [Accessed 10 Feb. 2018].

[10] Looperman. (2018). *Download Free Drum Samples*. [online] Available at: https://www.looperman.com/loops/cats/royalty-free-drum-loops-samples-sounds-wavs-download [Accessed 7 May 2018].

[11]The Official MIDI Specifications. (2014). *MIDI DIN Electrical Specification*. [online] Available at: https://www.midi.org/specifications/item/midi-din-electrical-specification [Accessed 13 Feb. 2018].

[12]En.wikipedia.org. (2018). *Line level*. [online] Available at: https://en.wikipedia.org/wiki/Line_level [Accessed 19 Feb. 2018].

[13]Learn.sparkfun.com. (2018). *Pull-up Resistors*. [online] Available at: https://learn.sparkfun.com/tutorials/pull-up-resistors [Accessed 28 May 2018].

[14]MSP-EXP432P401R Quick Start Guide. (2015). [PDF] Texas Instruments, p.1. Available at: http://www.ti.com/lit/ml/slau596a/slau596a.pdf [Accessed 16 Apr. 2018].

[15] G.P. Thomas, "What Materials Are Used in Drum Kits?," *azom.com*, para. 4, Nov. 23, 2012. [Online]. Available: https://www.azom.com/article.aspx?ArticleID=7889. [Accessed: Nov. 1, 2017].

[16] "IEEE Code of Ethics," [Online]. Available:https://www.ieee.org/about/corporate/governance/p7-8.html. [Accessed: Nov. 4, 2017].

[17] M. Nicholl, Introduction to MIDI/Synthesis. Miami, FL: CPP/Belwin, 1993.

[18] S. Skillings, "Systems for combining inputs from electronic musical instruments and devices," *USPTO Full-Text and Image Database*, 19 Sept. 2017. [Online]. Available: http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&u=%2Fnetahtml%2FPTO%2Fsearch-adv.htm&r=13&f=G&l=50&d=PTXT&p=1&S1=(((drum+AND+machine)+AND+analog)+AND+instrument)&OS=drum+AND+machine+AND+analog+AND+instrument&RS=(((drum+AND+machine)+AND+analog)+AND+instrument). [Accessed: Oct. 11, 2017].

[19] M. Garido, J. Grajal, M. Lopez-Vallejo and M. A. Sanchez. "Implementing FFT-based digital channelized receivers on FPGA platforms," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 44, no. 4, pp. 1567-1585, Oct. 2008. [Online]. Available: IEEE Xplore, http://ieeexplore.ieee.org/document/4667732/. [Accessed: Oct. 11, 2017].

[20] "Cmod A7 Reference Manual," June 24, 2016. [Online]. Available: https://reference.digilentinc.com/_media/cmod_a7/cmod_a7_rm.pdf. [Accessed: Oct. 11, 2017].

[21] I.S. Gibson, D.M. Howard, and A.M. Tyrell, "Real-time signing synthesis using a parallel processing system," in *Proceedings of the IEEE Colloquium on Audio and Music Technology: The Challenge of Creative DSP, Nov. 18, 1998, London, UK* [Online]. London: IET, 1998. Available: IEEE Xplore, http://ieeexplore.ieee.org/document/757353/. [Accessed: Oct. 12, 2017].

[22] R.H. Hosking, "How to Choose the Right FPGA," *eetimes.com*, Jan. 18, 2007. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1275359. [Accessed: Oct. 11, 2017].

[23] Y. Yamaguchi and S. Yoshida, "A study of an FPGA synthesizer," in *Proceedings of the 2010 International Conference on Audio and Image Processing, 23-25 Nov., 2010, Shanghai, China*[Online]. Available: IEEE Xplore, http://ieeexplore.ieee.org/document/5685161/. [Accessed: Oct. 11, 2017].

[24] Y. Atakhanian, "System, method and computer program product for generating musical notes via a user interface touchpad," *USPTO Full-Text and Image Database*, 29 Aug. 2017. [Online]. Available: http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&u=%2Fneta html%2FPTO%2Fsearch-adv.htm&r=10&f=G&l=50&d=PTXT&p=1&S1=(((electronic+AND+musical)+AND+instrument)+AND+percussion)&OS=electronic+AND+musical+AND+instrument+AND+percussion&RS=(((electronic+AND+musical)+AND+instrument)+AND+percussion). [Accessed: Oct. 11, 2017].

[25] G. Ning, J. Zhang, and S. Zhang, "Design of audio signal processing and display system based on SoC," in *Proceedings of the 4th International Conference on Computer Science and Network Technology, Dec. 19-20, 2015, Harbin, China* [Online]. Harbin, China: IEEE, 2015. Available: IEEE Xplore, http://ieeexplore.ieee.org/document/7490868/. [Accessed: Oct. 12, 2017].

[26] O. Gilliet and G. Richard, "Transcription and Separation of Drum Signals from Polyphonic Music," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 16, no. 3, p. 529-540, Feb. 2008. [Online]. Available: IEEE Xplore, http://ieeexplore.ieee.org/document/4443887/. [Accessed: Oct. 12, 2017].

[27] *IEEE Std 1233, 1998 Edition*, p. 4 (10/36), DOI: 10.1109/IEEESTD.1998.88826.

Appendix A – Senior Project Analysis

TABLE XIII
ANALYSIS OF SENIOR PROJECT DESIGN

Project Title:	Electronic Drum Machine (E.D.M.)
Student's Name:	Nyssa Backes
Advisor's Name:	Dr. Joseph Callenes-Sloan
Date:	November 11, 2017

1. Summary of Functional Requirements
Essentially, the EDM allows users to output electronic drum tracks to a speaker, computer, or a variety of other electronic instruments. The EDM allows users to select from a variety of pre-programmed tracks, or use the included library of 11 individual drum noises to program their own tracks. These tracks can utilize a variety of tempos and beats-per-measure (bpm) configurations. Furthermore, the EDM increases durability and portability by withstanding falls from short distances and operating using only battery power alone for at least 5 hours.

2. Primary Constraints
Configuring the EDM's memory currently poses the most difficulty. As the fundamental development choice revolves around whether digital or analog systems generate the individual drum noises, a great many methods of memory implementation exist. If employing a digital-based design, using an FPGA or a microcontroller to play back digital samples of the required drum noises, then memory implementation may require communication with an external memory module, using a specified communication protocol. Yet, if the EDM uses an analog design then a separate system must store both the user defined and pre-programmed tracks as analog circuitry simply generates noises without any intrinsic means of storing this information.
Furthermore, EDM customer needs were determined and translated into engineering specifications (see TABLE I). These specifications informed all subsequent design choices and provided benchmarks of project success.

3. Economic
The EDM could potentially put human drummers out of work, which causes an extremely negative impact on the economy. As previously mentioned in the abstract, purchasing a system such as the EDM serves as a low cost alternative to hiring a human drummer. While such a decision could positively impact the economy in many ways (for example, allowing musicians/producers without the financial means of hiring a drummer to develop music with drums, and this music may make them economically successful enough to afford a human drummer) there also exists the very real possibility that electronic drum machines, such as EDM, may take business away from actual human drummers as these machines do the same job for much less money.
Principally, the EDM provides economic opportunity for producer and musicians that lack the financial resources to hire a human drummer. Purchasing an EDM may afford producers and musicians many job opportunities which would not exist for them without either having a human drummer or having a drum machine.
Furthermore, the EDM requires some monetary investment on part of those who purchase a device, but this investment is much smaller than the investment required to hire a human drummer. For example, the EDM has a projected total cost of $250, while professional drummers can cost upwards of $400 per day [1].
The pre-manufactured individual components (namely various integrated circuits, printed circuit boards, FPGA, and others) comprise most of the manufactured capital costs associated with the EDM. Furthermore, future software and hardware updates require an additional investment of both manufactured and human capital. These updates include both expanded libraries of individual drum noises and additional pre-programmed drum tracks, as

well as expanded device memory. As well as requiring manufactured capital, the individual components used to construct the EDM (namely integrated circuit chips, printed circuit boards, discrete electrical components, and others) also require natural resources such as silicon, copper, and other heavy metals.

As with any project, potential costs and benefits of the EDM extend far beyond those deriving from its intended use. Such costs and benefits include potentially lowering the costs, both of production and sale, of music and concert tickets. These potentially reduced costs could derive from the lowered cost of producing and recording music, transporting equipment and personnel for tours, and purchasing new instruments. Purchasing and transporting an EDM costs significantly less than purchasing and transporting a full size drum set, as well as the costs incurred by employing and transporting a human drummer. Thus, with less financial overhead required to produce and record music, as well as tour, purchasing music and concert tickets should also become less expensive. Furthermore, manufacturing and future development of the EDM develops new jobs as people will be employed to develop and implement the EDM's future software and hardware updates.

As the EDM is currently in the very initial stages of design and development a complete parts list is not yet achievable. However, Table X details the expected costs of all required parts, shipping for these parts, and labor to complete the project. Currently, the EDM's principle requirement involves satisfying the requirements of my senior project, and thus, I provide all required funding.

Additional materials, beyond those directly needed for the EDM's prototyping and physical construction, may aid in the design and testing process. Cal Poly provides most of such materials (namely oscilloscopes, power supplies, breadboards, MATLAB, various cables and probes, and other such materials) for student use. Furthermore, other design and testing materials, such as LTSpice, PSpice, Fritzing, and Eagle CAD software packages, remain freely available to the public.

When considering the adaption of the EDM for commercial manufacture and sale, individual EDM units would likely retail for a projected $250 each, which places them competitively in a market where similar products retail for between $500 and $600 each [5]. Including the $250 per unit price, and the estimated $150 per unit manufacturing costs, signifies that each unit would net $100 profit. If such a manufacturing endeavor began immediately, EDM units would begin hitting store shelves in approximately 9 months, a timeline which includes a 6 month design period, with an additional 3 months to implement the means of commercial manufacturing. Furthermore, Figs. 3-5 of the accompanying report extensively detail the 6 months required for design in Gantt chart format.

4. If manufactured on a commercial basis:

The following break-even analysis considers the potential commercial manufacturing of the EDM.

Let n = total number of EDM units sold

Assume $10 million for fixed costs, like production facility rent, required insurance, purchase of production equipment, and others.

Total Cost = Fixed Costs + n*(variable cost per unit) = $10 million + n*($150)

Revenue = n*(sales price per unit) = n*($250)

The break-even point represents the number of units sold (n) required for Revenue to equal Total Cost.

$$10 \text{ million} + 150*n = 250*n$$
$$10 \text{ million} = 100*n$$
$$n = 100{,}000$$

Thus, after selling 100,000 EDM units, the total cost of production equals the revenue generated, and thus, n = 100,000 represents the break-even point. Furthermore, when any more than 100,000 EDM units are sold, the manufacturer profits. Thus, if an estimated 200,000 units are sold annually, the EDM generates a total yearly profit of $10 million.

5. Environmental

As with any other electronic device, the manufacture of the EDM poses certain risks to the environment. As electronic devices, the EDM included, contain materials such as silicon, lead, copper, and a variety of other heavy metals, their manufacture and eventual disposal release a significant amount of dangerous materials into the environment. However, the EDM positively affects the environment by reducing the number of drum sets manufactured annually. The manufacture of drum sets harms the environment in two specific ways. Firstly, the heads of drums consist of a material called polyethylene terephtlatae, a petroleum derivative whose manufacture requires significant fossil fuel use [15]. Additionally, the manufacture of drum sets involves a great amount of lumber, for the drum shells, which is not always responsibly sourced and can possibly contribute to the world-wide deforestation epidemic [15]. Additionally, the manufacture of drum sets involves a great amount of lumber, for the drum shells, which is not always responsibly sourced and can possibly contribute to the world-wide deforestation epidemic [15].

6. Manufacturability

System timing presents a significant potential manufacturing issue. Since nearly every function of the EDM relies on the proper timing of other functions to execute as expected, accurate timing is essential to proper device function. The fundamental need for accurate timing becomes a potential manufacturing issue when considering real-world propagation delays, especially in an analog system. Thus, without properly accounting for the inherent propagation delays of the physical system during manufacturing, the EDM may not meet any of its design requirements or engineering specifications.

Additionally, the assembly process presents specific difficulties as all of the EDM’s internal circuitry must fit inside its specifically-sized enclosure. Thus, wiring and connecting components must be done in such a way as to minimize required space, so that everything fits inside the EDM’s enclosure.

Lastly, the circuits employed to generate some of the E.D.M’s drum noises must be tested, following their construction, to ensure accuracy. Also, the output of these circuits should be measured carefully to ensure that the maximum peak-to-peak voltage output does not exceed the allowed peak-to-peak input voltage, and DC offset voltage, of the MSP432’s analog to digital converter.

7. Sustainability

The EDM requires both software updates to increase the variety of musical styles and genres that the EDM accommodates, as wells as frequent battery replacements since the EDM functions only on battery-power. This battery dependence impacts the environment significantly since, without using rechargeable batteries, the EDM contributes greatly to the amount of toxic waste improperly discarded into nature. As previously stated, future improvements to the EDM include software updates with expanded libraries of individual drum noises and pre-programmed drum tracks. Such software updates present no immediate implementation challenges.

8. Ethical

The EDM does not appear to commit any ethical violations, as described by the IEEE Code of Ethics [16]. To avoid potentially harming the public, proper documentation warning of any possible danger arising from use or misuse of individual devices accompanies the EDM during purchase. The accompanying report includes proper disclosure of all potential conflicts of interest, as well as citations for any outside resources consulted in designing the EDM. Furthermore, the accompanying report retains a quality of honesty and realistic expectation when listing and testing all requirements and specifications of the EDM. The EDM seeks to bring technology into the hands of musicians who may have never thought to use it in a way, as a means of improving the quality of their art and lives. Additionally, the accompanying report has withstood several rounds of peer and advisor review, all the while accumulating helpful technical feedback for improvement. Lastly, the overall EDM design considers users of all abilities and disabilities and seeks to include people of all varieties and experience levels, all while doing no harm to anyone. Thus, by the IEEE code of ethics, the EDM appears ethically designed and responsibly constructed.

The EDM commits no ethical violation, as described by the ethical principle of utilitarianism. As utilitarianism seeks to do the most good for the most people, it is understood that the EDM may cause economic harm to drummers as a means of economically benefitting others. Since more non-drummer musicians who could benefit from EDM exist than drummers who face negative impacts at its hands, by the tenets of utilitarianism, the EDM does not commit any ethical violations.

Potential ethical issues arise from acoustic output generated by the E.D.M. but, as the E.D.M. does not create sound itself (rather, it creates a MIDI or 3.5mm output signal that must be acoustically played by another device) these issues lie with the acoustic devices attached to the E.D.M., not the E.D.M. itself.

9. Health and Safety

While the design process does not present any immediately apparent health and safety concerns, some concerns do arise when considering the manufacturing process. As with many other electronic devices, the EDM contains

lead and other toxic heavy metals. Direct exposure to these dangerous materials during the manufacturing process poses significant health and safety risks. Furthermore, regular intended use, or even gross misuse, of the EDM does not pose any immediately apparent health and safety concerns. No health and safety concerns arise from these situations solely because the EDM does not output sound directly, which could cause hearing damage, but rather outputs audio data to an outside device, and relies only on regular consumer battery power. Lastly, EDM positively impacts health and safety by preventing many of the injuries commonly associated with drumming, By using an EDM, rather than playing a traditional drum set, users prevent, or at the very least substantially reduce their risk of, many injuries commonly incurred by playing drums such as fractured hands and fingers, hand bruises, minor cuts, and facial injuries (from flying drumsticks or broken cymbal pieces).
As previously mentioned, the E.D.M. does not produce acoustic output itself, rather, it produces a signal that must be played by another device, such as a speaker or an amplifier. Thus, any potential health and safety concerns arising from acoustic output lie with the output device (speaker, amplifier, etc.) and not the E.D.M.

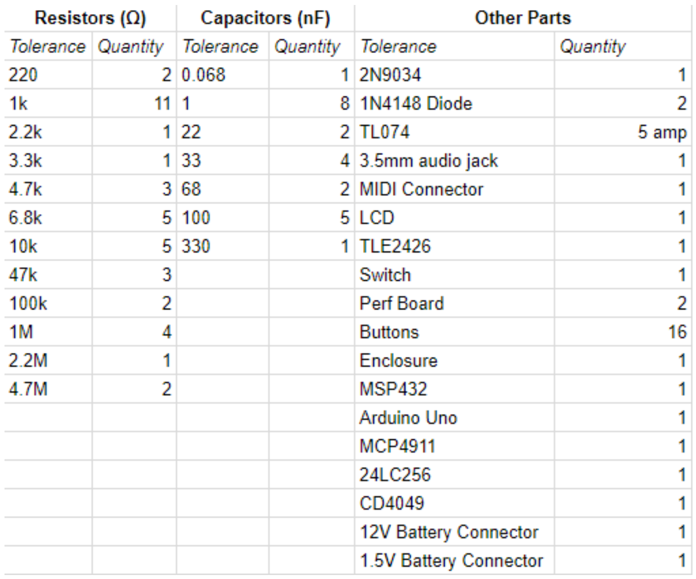
10. Social and Political
The EDM's direct stakeholders include musicians, music producers, people with no drumming experience, people with a multitude of drum experience, and drummers. Indirectly, the EDM influences nearly all people who purchase music or attend live music concerts as the EDM can potentially influence types of music produced, environment of live music performances, and the prices of all such items. Nearly all of the EDM's direct stakeholders benefit from the EDM, except drummers. As previously discussed, while the EDM may provide creative and economic opportunity to music producers and other types of Non-drummer musicians, the EDM could take jobs away from professional drummers. Furthermore, the EDM could potentially lead to people wanting to learn how to program drums rather than play them, which would ultimately reduce the number of drummers in the world. Thus, EDM creates an inequity in economic demand for drummers versus other types of musicians, which stands to positively impact other kinds of musicians, at the expense of drummers.

11. Development
During the design and development process I consulted many outside resources (see References section) to independently learn many skills needed to complete this project. Such skills include MIDI interfacing, PCB design, and various techniques of audio filtering and processing. Through intensive research, I found and studied a conceptual overview of MIDI and electronic music synthesis, which are subjects essential to the EDM's ability to interface with both a computer and other electronic instruments [17]. Further study afforded an insight into various methods of interfacing multiple electronic instruments and devices, using MIDI and other techniques [18]. Additionally, I explored many possible system designs, including systems utilizing FPGAs, like the CMOD A7 and others, to process signals using fast Fourier transforms (FFTs) or parallel processing techniques [19]-[22], [25]. To further inform future design choices regarding FPGAs, I consulted an in-depth analysis of a synthesizer built using an FPGA [23]. Furthermore, research concerning different methods that existing consumer devices use to generate musical notes via user interface informs the EDM's development greatly as the EDM serves this exact same purpose [24]. Lastly, I consulted a reference that discusses methods of separating and processing drum signals as part of polyphonic music, essentially providing information regarding the reverse engineering of the synthesizing and processing components of an electronic drum machine [26].

Appendix B – Parts List and Cost

The following table, Table XIV, provides a complete list of all of the parts used to complete the E.D.M.

TABLE XIV
COMPLETE PARTS LIST



Resistors (Ω)		Capacitors (nF)		Other Parts	
Tolerance	*Quantity*	*Tolerance*	*Quantity*	*Tolerance*	*Quantity*
220	2	0.068	1	2N9034	1
1k	11	1	8	1N4148 Diode	2
2.2k	1	22	2	TL074	5 amp
3.3k	1	33	4	3.5mm audio jack	1
4.7k	3	68	2	MIDI Connector	1
6.8k	5	100	5	LCD	1
10k	5	330	1	TLE2426	1
47k	3			Switch	1
100k	2			Perf Board	2
1M	4			Buttons	16
2.2M	1			Enclosure	1
4.7M	2			MSP432	1
				Arduino Uno	1
				MCP4911	1
				24LC256	1
				CD4049	1
				12V Battery Connector	1
				1.5V Battery Connector	1

Table XV, shown below, provides an initial cost estimate for the E.D.M. This estimate was determined before the competition of any design, construction, or testing.

TABLE XV
Estimated Project Costs

Item	Cost
Labor	$9,400.00
Parts	$217.00
Shipping, for parts	$32.00
Total Cost	$9,649.00

Labor cost estimate derives from an estimated 200 hour project duration, and a projected labor cost of $47.00 per hour. Estimated parts cost derives from the PERT model,

$$t_e = t_o + 4t_r + 6t_p$$

with an optimistic cost (t_o) of $150, a realistic cost ($t_r$) of $200, and a pessimistic cost (t_p) of $350. Additionally, the PERT model determined an estimation of shipping costs, with an optimistic (t_o) cost of $20, a realistic cost ($t_r$) of $30, and a pessimistic cost (t_p) of $50.
The PERT model derives from [6], Chapter 10.

Finally, Table XVI, shown below, provides a complete and final total cost for the E.D.M.

TABLE XVI
Complete Project Costs

Item	Cost
Labor	$9,400.00
Parts	$225.63
Shipping, for parts	$39.89
Total Cost	$9,665.52

Again, labor cost estimate derives from an estimated 200 hour project duration, and a projected labor cost of $47.00 per hour.
Thus, as the initial estimated cost was $9,649.00 and the final cost was $9,665.52, the overall project was completed $16.52, or 0.17%, over the original budget.

Appendix C – Project Schedule

The following charts (see Figs.45 - 47) provide an estimated timeline of project completion in Gantt format. These charts provide task lists, estimated times of completion, and all relevant completion milestones beginning in EE 460 (Fall 2017) and ending in EE 462 (Spring 2018).

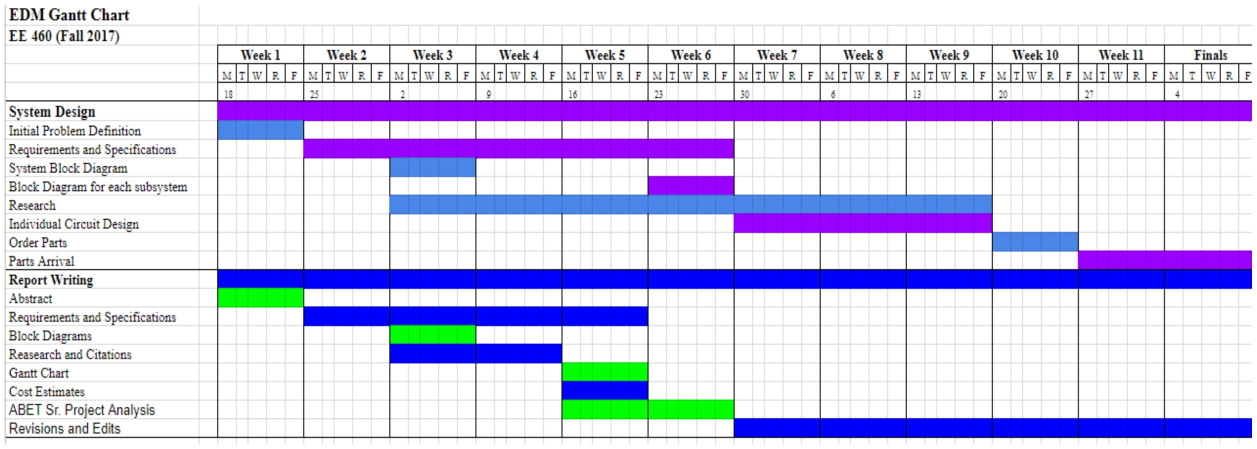


Figure 45 - The above Gantt chart was created for the duration of EE 460 (Fall 2017). As this Gantt chart was created during that quarter, it accurately reflects the project progress made during that time.

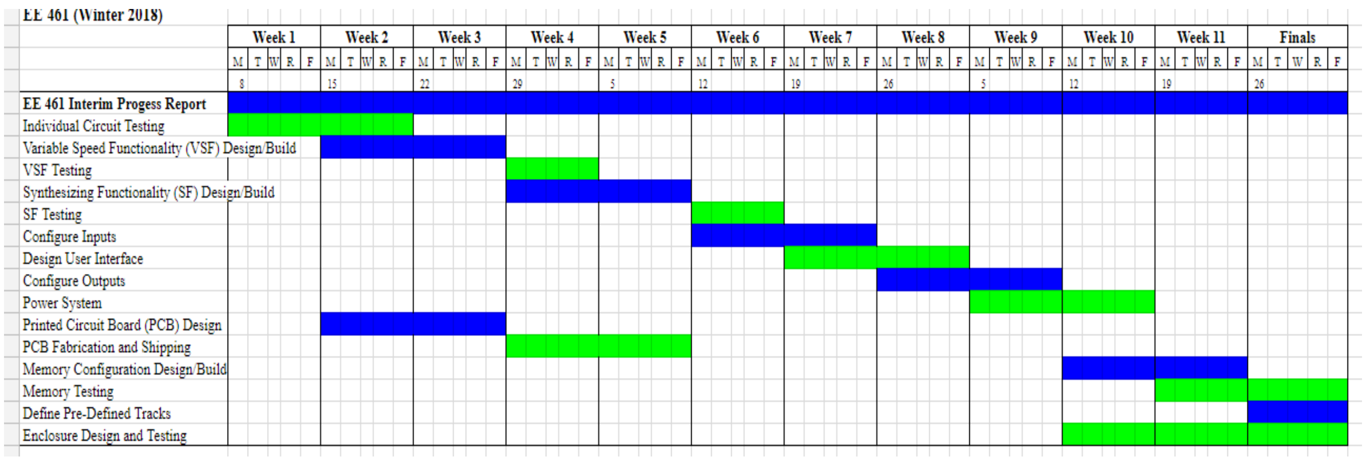


Figure 46 - The above Gantt chart reflects the project plan for EE 461 (Winter 2018).

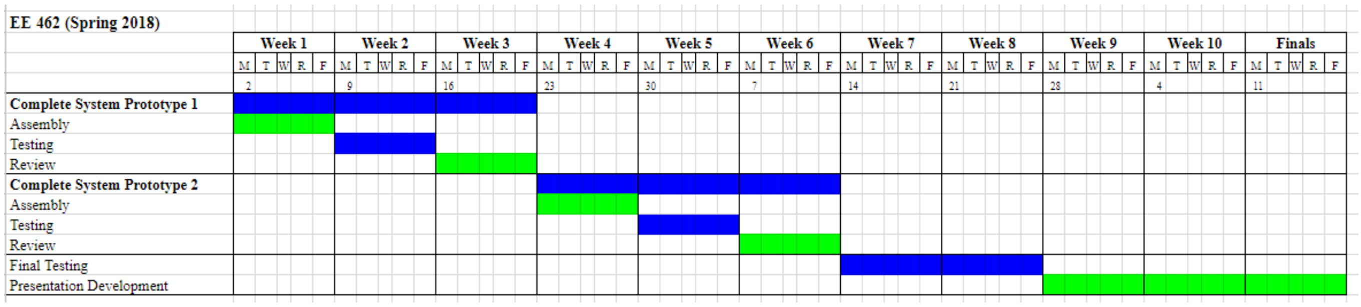


Figure 47 - This Gantt chart reflects the project plan for EE 462 (Spring 2018).

The following Gantt charts (see Figs. 48 and 49) reflect the actual weekly progress of the E.D.M. project over the course of two academic quarters (Winter 2018 and Spring 2018).

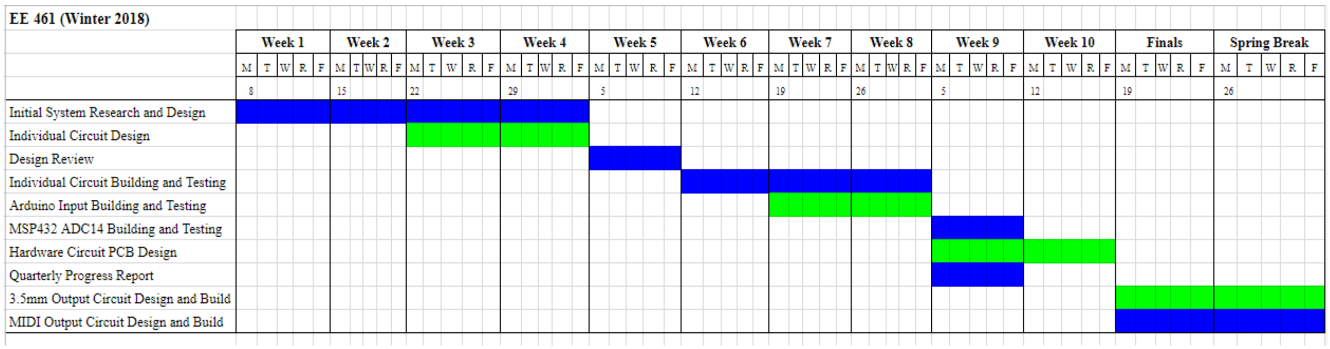


Figure 48 - This Gantt chart reflects the actual weekly progress made during EE 461 (Winter 2018).

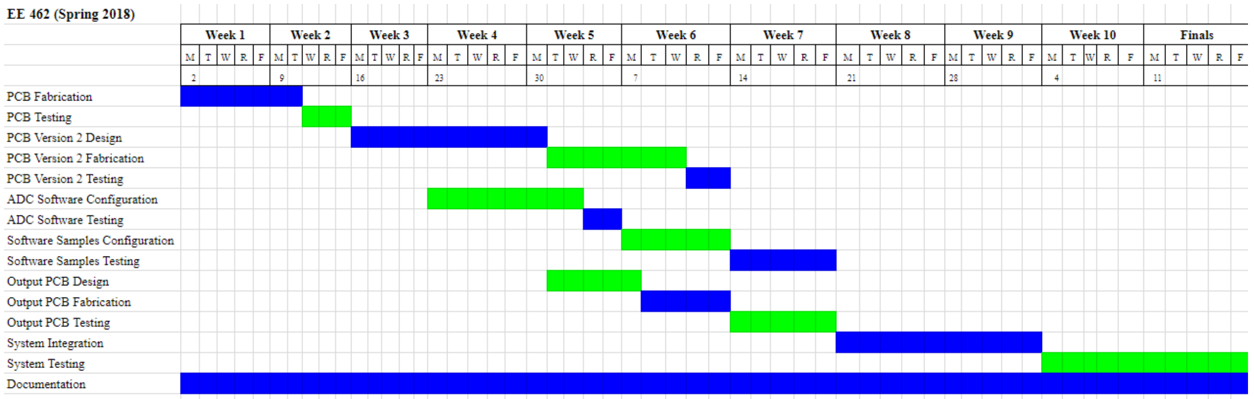


Figure 49 - The above Gantt chart shows the actual weekly progress made during EE 462 (Spring 2018).

Appendix D – Complete PCB Layouts

The following diagrams provide the layout of each of the 2 PCBs featured in the E.D.M.

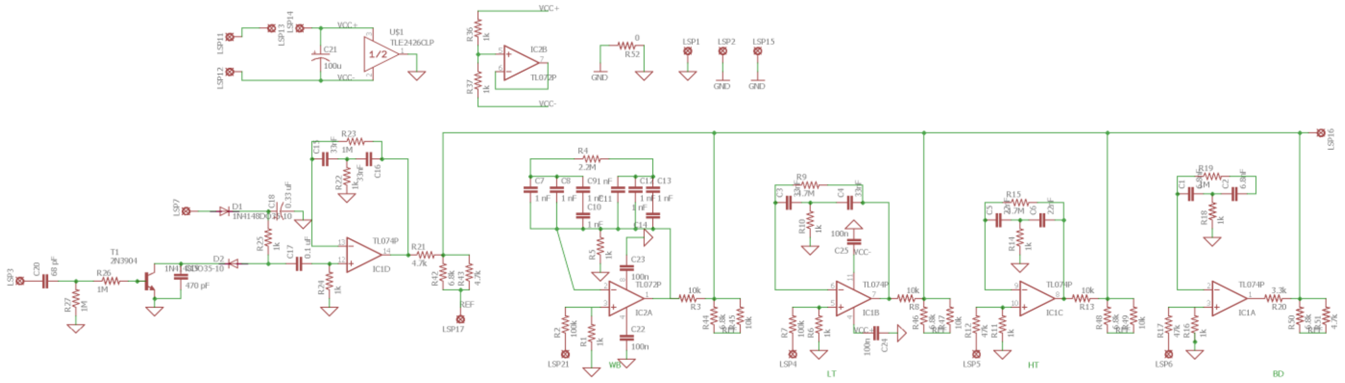


Figure 50 - The above PCB schematic includes all of the circuitry necessary for the creation of the hardware generated drum sounds.

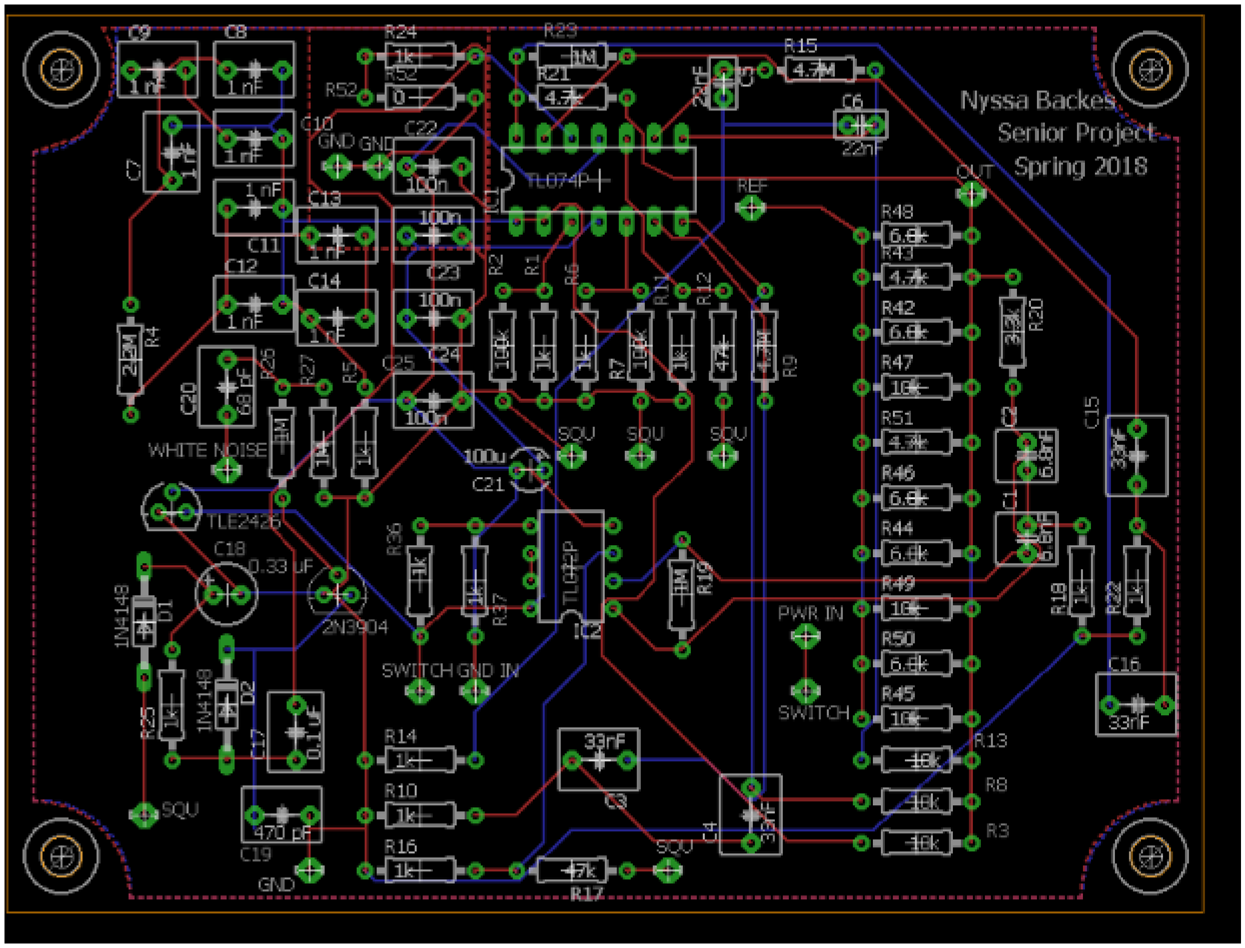


Figure 51 - The above PCB layout includes all of the circuitry necessary for the creation of the hardware generated drum sounds.

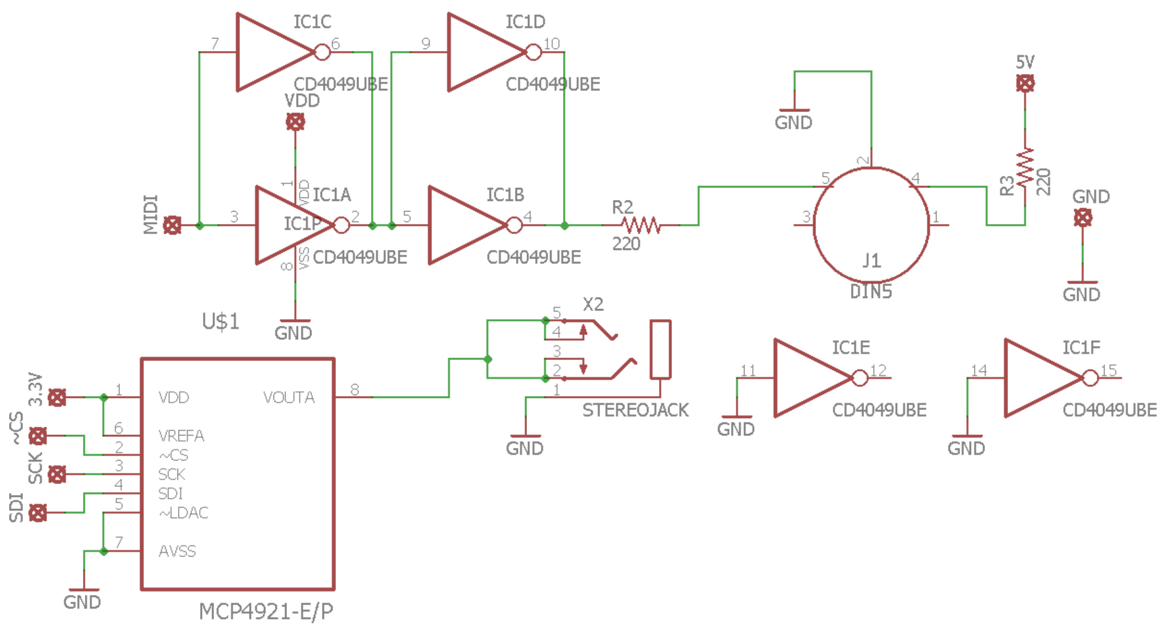


Figure 52 - The above PCB schematic includes all of the circuitry necessary for both the MIDI output functionality and the 3.5mm output functionality.

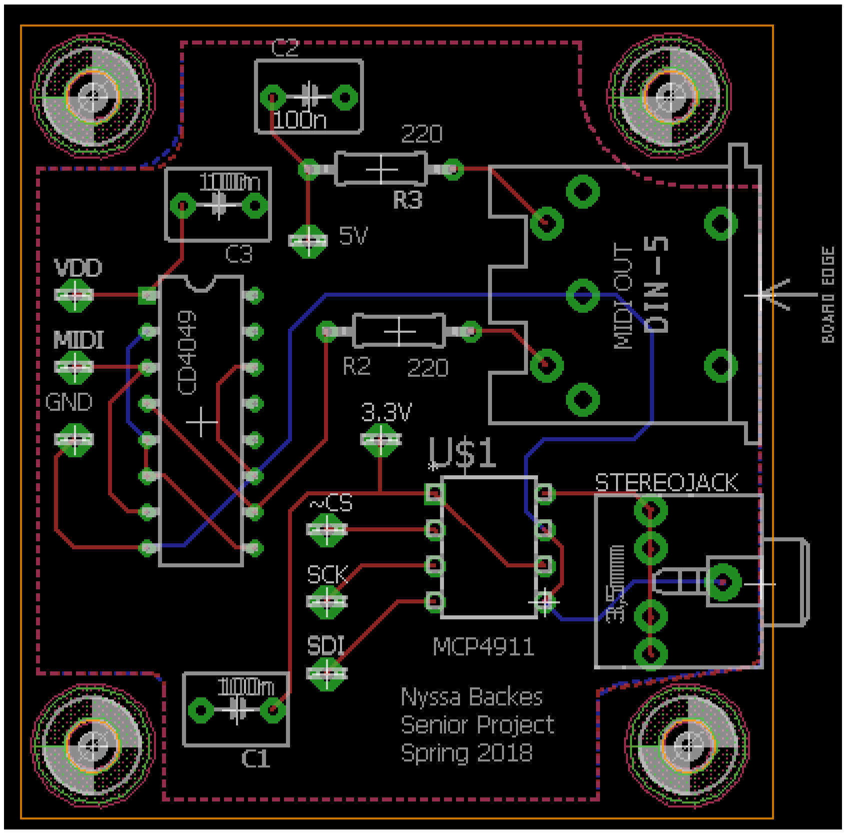


Figure 53 - The above PCB layout includes all of the circuitry necessary for both the MIDI output functionality and the 3.5mm output functionality.

Appendix E – Program Listing

The following listing includes all programs developed and employed for the design and operation of the E.D.M.

MATLAB Programs

convert_samples.m

```
%----------------------------------------------------------------
% Program that converts .WAV files to tables of bytes
% for saving in EEPROM. Also, this program allows resampling
% of audio to ensure appropriate resultant size.
%----------------------------------------------------------------
%read .WAV file data into array (y) and return original
%sampling rate (Fs)
[y, Fs] = audioread('file_name.wav');
%resmaples audio data in y at (100/441)*Fs
%the values provided (11 and 441) are arbitrary in this instance
y_new = resample(y,100,441);
a = size(y_new);
new = [];
output = [];
final = [];
i = 1;
j = 1;
k = 1;
%find maximum and minimum values of data for scale correction
[find_extrema, Fs] = audioread('file_name.wav', 'native');
y_max = max(find_extrema);
y_min = min(find_extrema);

%scale data (-1 to 1)
while i <= a(1)
    b = y_last(i);
    c = (b + 1)*[(y_max - y_min)/2] + y_min;
    new = vertcat(new, c);
    i = i + 1;
end

%convert values to int16
 while j <= a(1)
     b = new(j);
     c = int16(b);
     output = vertcat(output, c);
     j = j + 1;
 end



 while k < a(1)
    d = output(k);
    f = dec2bin(abs(d));
    z = size(f);
    if (z(2) ~= 16)
        x = zeros(1, (16 - z(2)));
        f = horzcat(x, f);
    end
    final = vertcat(final, f);
    k = k + 1;
 end
```

Arduino Programs

COMPLETE_ARDUINO

```
/* This program controls the generation of square waves and white noise
   pulses to drive each of the hardware-generated drum circuits. */

// pin setup
const int SDSquareButton = 2;
const int WBSquareButton = 3;
const int LTSquareButton = 4;
const int HTSquareButton = 5;
const int BDSquareButton = 6;
const int BDSquarePin = 7;
const int HTSquarePin = 9;
const int LTSquarePin = 10;
const int WBSquarePin = 11;
const int SDSquarePin = 12;
const int whiteNoisePin = 13;

// other variables
boolean WNnoiseOn = false;
boolean SDsquareOn = false;
boolean WBsquareOn = false;
boolean LTsquareOn = false;
boolean HTsquareOn = false;
boolean BDsquareOn = false;
unsigned long startTime = 0;
unsigned long currentTime = 0;
const long debounce = 200;   // for debouncing of buttons
const unsigned long period = 300;  //miliseconds until square wave turns off
unsigned long int reg;       // for white noise generation
int SDsquareState = 0;
int WBsquareState = 0;
int LTsquareState = 0;
int HTsquareState = 0;
int BDsquareState = 0;

void setup() {
  pinMode(BDSquarePin, OUTPUT);
  pinMode(HTSquarePin, OUTPUT);
  pinMode(LTSquarePin, OUTPUT);
  pinMode(WBSquarePin, OUTPUT);
  pinMode(SDSquarePin, OUTPUT);
  pinMode(whiteNoisePin, OUTPUT);
  pinMode(BDSquareButton, INPUT_PULLUP);
  pinMode(HTSquareButton, INPUT_PULLUP);
  pinMode(LTSquareButton, INPUT_PULLUP);
  pinMode(WBSquareButton, INPUT_PULLUP);
  pinMode(SDSquareButton, INPUT_PULLUP);

  // Arbitrary inital value (seed for white noisebitstream); must not be zero
  reg = 0x55aa55aaL;
```

```
}

void loop() {
   currentTime = millis();

   //SD stuff
   SDreadButton();
   SDsquareWaveOn();
   checkTime();

   //LT Stuff
   LTreadButton();
   LTsquareWaveOn();
   checkTime();

   //WB stuff
   WBreadButton();
   WBsquareWaveOn();
   checkTime();

   //HT stuff
   HTreadButton();
   HTsquareWaveOn();
   checkTime();

   //BD stuff
   BDreadButton();
   BDsquareWaveOn();
   checkTime();

}

void SDreadButton() {

  // check for button pushed (on/off control)
  boolean SDbuttonPushed = (digitalRead(SDSquareButton) == LOW);

  if (SDbuttonPushed && millis() - startTime > debounce) {
    if (digitalRead(SDSquareButton) == LOW) {
      SDsquareOn = !SDsquareOn;
      WNnoiseOn = !WNnoiseOn;
      startTime = millis();
    }
  }
}

void LTreadButton() {

  // check for button pushed (on/off control)
  boolean LTbuttonPushed = (digitalRead(LTSquareButton) == LOW);

  if (LTbuttonPushed && millis() - startTime > debounce) {
    if (digitalRead(LTSquareButton) == LOW) {
      LTsquareOn = !LTsquareOn;
      startTime = millis();
    }
  }
}

void WBreadButton() {

  // check for button pushed (on/off control)
  boolean WBbuttonPushed = (digitalRead(WBSquareButton) == LOW);

  if (WBbuttonPushed && millis() - startTime > debounce) {
    if (digitalRead(WBSquareButton) == LOW) {
      WBsquareOn = !WBsquareOn;
      startTime = millis();
    }
  }
}

void HTreadButton() {

  // check for button pushed (on/off control)
  boolean HTbuttonPushed = (digitalRead(HTSquareButton) == LOW);

  if (HTbuttonPushed && millis() - startTime > debounce) {
    if (digitalRead(HTSquareButton) == LOW) {
      HTsquareOn = !HTsquareOn;
      startTime = millis();
    }
  }
}

void BDreadButton() {

  // check for button pushed (on/off control)
  boolean BDbuttonPushed = (digitalRead(BDSquareButton) == LOW);

  if (BDbuttonPushed && millis() - startTime > debounce) {
    if (digitalRead(BDSquareButton) == LOW) {
      BDsquareOn = !BDsquareOn;
      startTime = millis();
    }
  }
}

void SDsquareWaveOn() {
  unsigned long int newr;             // new register value
  unsigned char b31, b29, b25, b24;  // 4 bits from 32-bit register
  unsigned char lobit;                // XOR 4 bits register together

  // Extract four bits from the 32-bit register
  b31 = (reg & (1L << 31)) >> 31;
  b29 = (reg & (1L << 29)) >> 29;
  b25 = (reg & (1L << 25)) >> 25;
  b24 = (reg & (1L << 24)) >> 24;
  // XOR the four bits together
  lobit = b31 ^ b29 ^ b25 ^ b24;

  // Shift and incorporate new bit at 0 bit position
  newr = (reg << 1) | lobit;

  // Replace register with new value
  reg = newr;

    if (SDsquareOn == true) {
      digitalWrite(whiteNoisePin, reg & 1);
        if(SDsquareState == 0){
            digitalWrite(SDSquarePin, LOW);
            SDsquareState = 1;
        }else{
            digitalWrite(SDSquarePin, HIGH);
            SDsquareState = 0;
        }
    }
    else {
        digitalWrite(SDSquarePin, LOW); // make sure it is off
        digitalWrite(whiteNoisePin, LOW);
    }
}

void WBsquareWaveOn() {
    if (WBsquareOn == true) {
        if(WBsquareState == 0){
            digitalWrite(WBSquarePin, LOW);
            WBsquareState = 1;
        }else{
            digitalWrite(WBSquarePin, HIGH);
            WBsquareState = 0;
        }
    }
    else {
        digitalWrite(WBSquarePin, LOW); // make sure it is off
    }
}

void LTsquareWaveOn() {
    if (LTsquareOn == true) {
        if(LTsquareState == 0){
            digitalWrite(LTSquarePin, LOW);
            LTsquareState = 1;
        }else{
            digitalWrite(LTSquarePin, HIGH);
            LTsquareState = 0;
        }
    }
    else {
        digitalWrite(LTSquarePin, LOW); // make sure it is off
    }
}
```

```
void HTsquareWaveOn() {
    if (HTsquareOn == true) {
        if(HTsquareState == 0){
            digitalWrite(HTSquarePin, LOW);
            HTsquareState = 1;
        }else{
            digitalWrite(HTSquarePin, HIGH);
            HTsquareState = 0;
        }
    }
    else {
        digitalWrite(HTSquarePin, LOW); // make sure it is off
    }
}

void BDsquareWaveOn() {
    if (BDsquareOn == true) {
        if(BDsquareState == 0){
            digitalWrite(BDSquarePin, LOW);
            BDsquareState = 1;
        }else{
            digitalWrite(BDSquarePin, HIGH);
            BDsquareState = 0;
        }
    }
    else {
        digitalWrite(BDSquarePin, LOW); // make sure it is off
    }
}

// check if 300ms have passed since button press
void checkTime() {
    if((currentTime - startTime)>= period){
        SDsquareOn = false;
        WNnoiseOn = false;
        WBsquareOn = false;
        LTsquareOn = false;
        HTsquareOn = false;
        BDsquareOn = false;
    }
}
```

MSP432 Programs

ADC_TEST.c

```
//****************************************************************************
// Test ADC by printing to terminal via UART. Uncalibrated.
//****************************************************************************

#include "msp.h"
#include "mytiming.h"

int flag = 0;
volatile unsigned int var = 0;
int millivolt = 0;
```

```
void UART0_init(void);
void delayMs(int n, int freq);
void setFreq(int freq);

int main(void) {

    setFreq(FREQ_3_MHZ);

    int mv_char_0 = 0;
    int mv_char_1 = 0;
    int mv_char_2 = 0;
    int mv_char_3 = 0;
    char uart_0;
    char uart_1;
    char uart_2;
    char uart_3;

    WDT_A->CTL = WDT_A_CTL_PW |              // Stop WDT
                 WDT_A_CTL_HOLD;

    // GPIO Setup
    P1->OUT &= ~BIT0;                        // Clear LED to start
    P1->DIR |= BIT0;                         // Set P1.0/LED to output
    P5->SEL1 |= BIT4;                        // Configure P5.4 for ADC
    P5->SEL0 |= BIT4;

    UART0_init();

    // Enable global interrupt
    __enable_irq();

    // Enable ADC interrupt in NVIC module
    NVIC->ISER[0] = 1 << ((ADC14_IRQn) & 31);

    // Sampling time, S&H=16, ADC14 on
    ADC14->CTL0 = ADC14_CTL0_SHT0_2 | ADC14_CTL0_SHP | ADC14_CTL0_ON;
    ADC14->CTL1 = ADC14_CTL1_RES_2;
    ADC14->MCTL[0] |= ADC14_MCTLN_INCH_1;   /* A1 ADC input select;
                                               Vref=AVCC */
    ADC14->IER0 |= ADC14_IER0_IE0;          /* Enable ADC conv
                                               complete interrupt */

    SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;  /* Wake up on exit from
                                              ISR */

    while (1)
    {
        ADC14->CTL0 |= ADC14_CTL0_ENC | ADC14_CTL0_SC;
        // Start sampling/conversion
        if (flag) {
            millivolt = (int) var;
            flag = 0;
            mv_char_3 = millivolt / 1000;
            mv_char_2 = millivolt / 100 - (mv_char_3 * 10);
            mv_char_1 = millivolt / 10 - (mv_char_3 * 100 + mv_char_2 * 10);
            mv_char_0 = millivolt - (mv_char_1 * 10 + mv_char_3 * 1000 +
                        mv_char_2 * 100);
            uart_0 = (char) mv_char_0 + '0';
            uart_1 = (char) mv_char_1 + '0';
            uart_2 = (char) mv_char_2 + '0';
            uart_3 = (char) mv_char_3 + '0';
            while(!(EUSCI_A0->IFG & 0x02)) { }
            EUSCI_A0->TXBUF = uart_3;
            while(!(EUSCI_A0->IFG & 0x02)) { }
            EUSCI_A0->TXBUF = uart_2;
            while(!(EUSCI_A0->IFG & 0x02)) { }
            EUSCI_A0->TXBUF = uart_1;
            while(!(EUSCI_A0->IFG & 0x02)) { }
            EUSCI_A0->TXBUF = uart_0;
            while(!(EUSCI_A0->IFG & 0x02)) { }
            EUSCI_A0->TXBUF = 0x0D;
            ADC14->CTL0 |= ADC14_CTL0_ENC | ADC14_CTL0_SC;
            P1->OUT &= ~BIT0;
        }
        else {
        __sleep();

        __no_operation();                   // For debugger
        }
    }
}

// ADC14 interrupt service routine
void ADC14_IRQHandler(void) {
    P1->OUT |= BIT0;
    var = ADC14->MEM[0];
    flag = 1;
}

void UART0_init(void)
{
    EUSCI_A0->CTLW0 |= 1;       // put in reset mode for config
    EUSCI_A0->MCTLW = 0;        // disable oversampling
    EUSCI_A0->CTLW0 = 0x0081;   // 1 stop bit, no parity,SMCLK,byte
                                // data
    EUSCI_A0->BRW = 26;         // 3,000,000 / 115200 = 26
    P1->SEL0 |= 0x0C;           // P1.3, P1.2 for UART
    P1->SEL1 &= ~0x0C;
    EUSCI_A0->CTLW0 &= ~1;      // take UART out of reset mode
}
```

ADC_Button_Test.c

```
//****************************************************************************
// Test ADC ON by external button and OFF by Timer A
//***************************************************************************

#include "msp.h"
#include "mytiming.h"

volatile unsigned int timer_flag = 0;
volatile unsigned int  button_flag_3 = 0;

#define   Num_of_Results   16
volatile uint16_t SDresults[Num_of_Results];
volatile uint16_t WBresults[Num_of_Results];
volatile uint16_t LTresults[Num_of_Results];
volatile uint16_t HTresults[Num_of_Results];
volatile uint16_t BDresults[Num_of_Results];
volatile uint8_t index1 = 0;
volatile uint8_t index2 = 0;
volatile uint8_t index3 = 0;
volatile uint8_t index4 = 0;
volatile uint8_t index5 = 0;
volatile int saved_IFG_3;

void setFreq(int freq);
void port_3_setup(unsigned char pinmask);

int main(void) {

    setFreq(FREQ_3_MHZ);

    WDT_A->CTL = WDT_A_CTL_PW |             // Stop WDT
                 WDT_A_CTL_HOLD;

    // GPIO Setup
    P6->SEL1 |= BIT0;                       // Configure P6.0 for ADC
    P6->SEL0 |= BIT0;

    // Initialize Port 3 (button connections)
    port_3_setup(BIT0);
    port_3_setup(BIT7);
    port_3_setup(BIT3);
    port_3_setup(BIT5);
    port_3_setup(BIT6);

    // Enable global interrupt
    __enable_irq();

    // Enable ADC and Timer A0 interrupts in NVIC module
    NVIC_EnableIRQ(TA0_0_IRQn);
    NVIC_EnableIRQ(ADC14_IRQn);
    NVIC_EnableIRQ(PORT3_IRQn);

    // Sampling time, S&H=16, ADC14 on
    ADC14->CTL0 = ADC14_CTL0_SHT0_2 | ADC14_CTL0_SHP | ADC14_CTL0_ON;
    ADC14->CTL1 = ADC14_CTL1_RES_2;
    ADC14->MCTL[0] |= ADC14_MCTLN_INCH_1; // A1 ADC input select; Vref=AVCC
    ADC14->IER0 |= ADC14_IER0_IE0;        // Enable interrupt (conv complete)

    SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;  // Wake up on exit from ISR
     while (1){
        if (button_flag_3){                       //button pushed, turn ADC on
            ADC14->CTL0 |= ADC14_CTL0_ENC | ADC14_CTL0_SC;
            //start sampling/conversion
            button_flag_3 = 0;                  //Reset flag
            P3->IFG = 0;                        //clear any pending flags
        }
        if (timer_flag){                        //time up, turn ADC off
            ADC14->CTL0 &= ~ADC14_CTL0_ENC;     //turn off ADC
            timer_flag = 0;
            P3->IE |= BIT0;                     //re-enable interrupts
            P3->IE |= BIT7;
            P3->IE |= BIT3;
            P3->IE |= BIT5;
            P3->IE |= BIT6;
        }
    }
}

void ADC14_IRQHandler(void) {   // ADC14 interrupt service routine
   if(saved_IFG_3 & BIT0){
       SDresults[index1] = ADC14->MEM[0];   //move results, IFG is cleared
       index1 = (index1 + 1) & 0x0F;
    }else if(saved_IFG_3 & BIT7){
       WBresults[index2] = ADC14->MEM[0];
       index2 = (index2 + 1) & 0x0F;
    }else if(saved_IFG_3 & BIT3){
       LTresults[index3] = ADC14->MEM[0];
       index3 = (index3 + 1) & 0x0F;
    }else if(saved_IFG_3 & BIT5){
       HTresults[index4] = ADC14->MEM[0];
       index4 = (index4 + 1) & 0x0F;
    }else if(saved_IFG_3 & BIT6){
       BDresults[index5] = ADC14->MEM[0];
       index5 = (index5 + 1) & 0x0F;
    }
    index = (index + 1) & 0x0F;          //increment results index, modulo
}

//Timer A0 interrupt service routine
void TA0_0_IRQHandler(void){
    timer_flag = 1;         //flag when timer reaches value
    TA0CCTL0 &= ~CCIFG;     //clear pending interrupt flag
    TA0CTL = 0;             //turn off the timer
}

void PORT3_IRQHandler(void)         //Interrupt handler for Port 3
{
       saved_IFG_3 = P3->IFG;
       button_flag_3 = 1;           //Set flag (button pushed)
       //configure Timer A0
       TA0CCR0 = 900000;            //Timer length = 300ms
       TA0CCTL0 |= CCIE;
       TA0CTL |= TASSEL_2 | MC_1;
       P3->IFG = 0;                 //Clear P3.0 pending interrupt flag
       P3->IE  &= ~BIT0;            //Disable interrupt for debouncing
       P3->IE  &= ~BIT7;          //Disable interrupt for debouncing
       P3->IE  &= ~BIT3;
       P3->IE  &= ~BIT5;
       P3->IE  &= ~BIT6;
}

void port_3_setup(unsigned char pinmask)
{
    P3->SEL0 &= ~pinmask;    // GPIO, not alternate function
    P3->SEL1 &= ~pinmask;
    P3->DIR  &= ~pinmask;    // input
    P3->OUT  |=  pinmask;    // pull up, not down
    P3->REN  |=  pinmask;    // enable pullup
    P3->IES  |=  pinmask;    // high->low transition
    P3->IFG  &= ~pinmask;    // clear possible stale IFG
    P3->IE   |=  pinmask;    // enable IFG
    return;
}
```

<u>EEPROM_Test.c</u>

```
//***************************************************************************
// Test an external button to access EEPROM. Display results on terminal via
// UART.
//***************************************************************************

#include "msp.h"
#include "mytiming.h"
#include "stdint.h"

void setFreq(int freq);
void delayMs(int n, int freq);
void port_4_setup(unsigned char pinmask);
void InitEEPROM(uint8_t DeviceAddress)
uint8_t ReadEEPROM(uint16_t MemAddress);
void UART0_init(void);

volatile int  button_flag = 0;                 //Flag to signal button press
volatile unsigned int UART_flag = 0;
uint16_t MemAddress;
uint16_t TransmitFlag = 0;
uint8_t ReceiveByte;
volatile unsigned int  button_flag = 0;
char check_1;
char check_2;
char check_3;
char check_4;
char check_5;
char check_6;
char check_7;
char check_8;
char check_9;
char check_10;
char check_11;
char check_12;
char check_13;
char check_14;
char check_15;
char check_16;
char check_17;
char check_18;
char check_19;
char check_20;
char check_21;
char check_22;
char check_23;
char check_24;
char check_25;

#define EEPROM_ADDRESS 0x50   //address of serial EEPROM

void main(void)
{
    setFreq(FREQ_3_MHZ);

    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer

    //Initialize Port 4 (button connection)
    port_4_setup(BIT0);

    // Enable global interrupt
    __enable_irq();

    // Enable interrupt in NVIC module
    NVIC_EnableIRQ(PORT4_IRQn);

    InitEEPROM(EEPROM_ADDRESS);    //initialize EEPROM

    UART0_init();                  //initialize UART

     /* The addresses in the example code below were used to check the ride
        cymbal. */
    while(1){                      //Endless loop
       if(button_flag == 1){       //button has been pressed
           MemAddress = 0x0641;                            //Address 1
           check_1 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_1;
           MemAddress = 0x05F2;                           //Address 2
           Check_2 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_2;
           MemAddress = 0x05A7;                           //Address 3
           Check_3 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_3;
           MemAddress = 0x0592;                           //Address 4
           check_4 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_4;
           MemAddress = 0x064C;                           //Address 5
           check_5 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_5;
           MemAddress = 0x059F;                          //Address 6
           check_6 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_6;
           MemAddress = 0x069B;                          //Address 7
           check_7 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_7;
           MemAddress = 0x05A1;                          //Address 8
           check_8 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_8;
           MemAddress = 0x05C6;                          //Address 9
           check_9 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_9;
           MemAddress = 0x06E0;                          //Address 10
           check_10 = (char) ReadEEPROM(MemAddress);
           while(!(EUSCI_A0->IFG & 0x02)) { }
           EUSCI_A0->TXBUF = check_10;

//*****************************************************************************
// While 25 addresses were checked during development, the code above only
// checks 10, to avoid unnecessary tedium in this report.
//*****************************************************************************
          }
           button_flag = 0;       //Reset flag
           P4->IE |= BIT0;        //re-enable interrupt
       }
     }
 }

void PORT4_IRQHandler(void)       //Interrupt handler for Port 4
{
    button_flag = 1;              //Set flag to signal button press detected
    P4->IFG = 0;                  //Clear pending interrupt flag
    P4->IE  &= ~BIT0;             //Disable interrupt for debouncing
}

void port_4_setup(unsigned char pinmask)
{
    P4->SEL0 &= ~pinmask;    // GPIO, not alternate function
    P4->SEL1 &= ~pinmask;
    P4->DIR  &= ~pinmask;    // input
    P4->OUT  |=  pinmask;    // pull up, not down
    P4->REN  |=  pinmask;    // enable pullup
    P4->IES  |=  pinmask;    // high->low transition
    P4->IFG  &= ~pinmask;    // clear possible stale IFG
    P4->IE   |=  pinmask;    // enable IFG
    return;
}

void UART0_init(void)
{
    EUSCI_A0->CTLW0 |= 1;       // put in reset mode for config
    EUSCI_A0->MCTLW = 0;        // disable oversampling
    EUSCI_A0->CTLW0 = 0x0081;   // 1 stop bit, no parity,SMCLK,byte
                                // data
    EUSCI_A0->BRW = 26;          // 3,000,000 / 115200 = 26
    P1->SEL0 |= 0x0C;            // P1.3, P1.2 for UART
    P1->SEL1 &= ~0x0C;
    EUSCI_A0->CTLW0 &= ~1;       // take UART out of reset mode
}

uint8_t ReadEEPROM(uint16_t MemAddress){

   //uint8_t ReceiveByte; MAYBE DON'T NEED THIS HERE
   uint8_t HiAddress;
   uint8_t LoAddress;

   HiAddress = MemAddress >> 8;
   LoAddress = MemAddress & 0xFF;

   EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TR;      //Set transmit mode (write)
   EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;   //I2C start condition

   while (!TransmitFlag);           //Wait for EEPROM address to transmit
   TransmitFlag = 0;

   EUSCI_B0 -> TXBUF = HiAddress;  //Send the high byte of the memory address

   while (!TransmitFlag);          //Wait for the transmit to complete
   TransmitFlag = 0;

   EUSCI_B0 -> TXBUF = LoAddress;  //Send the low byte of the memory address

   while (!TransmitFlag);          //Wait for the transmit to complete
   TransmitFlag = 0;

   EUSCI_B0->CTLW0 &= ~EUSCI_B_CTLW0_TR;   //Set receive mode (read)
   EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT; //I2C start condition (restart)

   // Wait for start to be transmitted
   while ((EUSCI_B0->CTLW0 & EUSCI_B_CTLW0_TXSTT));

   // set stop bit to trigger after first byte
   EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTP;

   while (!TransmitFlag);               //Wait to receive a byte
      TransmitFlag = 0;
      ReceiveByte = EUSCI_B0->RXBUF;    //Read byte from the buffer
   return ReceiveByte;
}

// Initialize I2C bus for communicating with EEPROM
void InitEEPROM(uint8_t DeviceAddress){

   P1->SEL0 |= BIT6 | BIT7;                //Set I2C pins of eUSCI_B0

   //Enable eUSCIB0 interrupt in NVIC module
   NVIC->ISER[0] = 1 << ((EUSCIB0_IRQn) & 31);
   //Configure USCI_B0 for I2C mode
   EUSCI_B0->CTLW0 |= EUSCI_A_CTLW0_SWRST;   //Software reset enabled
   EUSCI_B0->CTLW0 = EUSCI_A_CTLW0_SWRST |   //Remain eUSCI in reset mode
           EUSCI_B_CTLW0_MODE_3 |            //I2C mode
           EUSCI_B_CTLW0_MST |               //Master mode
           EUSCI_B_CTLW0_SYNC |              //Sync mode
           EUSCI_B_CTLW0_SSEL__SMCLK;        //SMCLK

   EUSCI_B0->BRW = 30;                       //baudrate = SMCLK / 30 = 100kHz
   EUSCI_B0->I2CSA = DeviceAddress;          //Slave address
   EUSCI_B0->CTLW0 &= ~EUSCI_A_CTLW0_SWRST;  //Release eUSCI from reset

   EUSCI_B0->IE |= EUSCI_A_IE_RXIE |         //Enable receive interrupt
                   EUSCI_A_IE_TXIE;
}

// I2C Interrupt Service Routine
void EUSCIB0_IRQHandler(void){

   if (EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG0){      //Check if transmit complete

         EUSCI_B0->IFG &= ~ EUSCI_B_IFG_TXIFG0;  //Clear interrupt flag
         TransmitFlag = 1;                       //Set global flag
   }

   if (EUSCI_B0->IFG & EUSCI_B_IFG_RXIFG0){      //Check if receive complete

         EUSCI_B0->IFG &= ~ EUSCI_B_IFG_RXIFG0;  //Clear interrupt flag
         TransmitFlag = 1;                       //Set global flag
    }
}
```

<u>DAC_Test.c</u>

```
//*****************************************************************************
// Test the functionality of the DAC chip.
//*****************************************************************************

#include "DriveDAC.h"

void SPI_init(void){
    // Configure port bits for SPI
    P4->DIR |= BIT1;           // P4.1 = /CS on the DAC
    P6SEL0 |= BIT6 + BIT7;     // P6.6 and P6.7 for UCB3SIM1 and UCB3CLK
    P6SEL1 &= ~(BIT6 + BIT7);

    // SPI Setup
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_SWRST;  // Put eUSCI state machine in
                                             // reset

    EUSCI_B0->CTLW0 = EUSCI_B_CTLW0_SWRST  // Remain eUSCI state machine in
                                           // reset
                      EUSCI_B_CTLW0_MST    // Set as SPI master
                      EUSCI_B_CTLW0_SYNC   // Set as synchronous mode
                      EUSCI_B_CTLW0_CKPL   // Set clock polarity high
                      EUSCI_B_CTLW0_MSB;   // MSB first

    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_SSEL__SMCLK; // SMCLK
    EUSCI_B0->BRW = 0x01;             // divide by 16, clock = fBRCLK/(UCBRx)

    // Initialize USCI state machine, SPI
    EUSCI_B0->CTLWO &= ~EUSCI_B_CTLW0_SWRST;
    // now waiting for something to be placed in TXBUF
    EUSCI_B0->IFG |= EUSCI_B_IFG_TXIFG;  // Clear TXIFG flag
}

void Drive_DAC(unsigned int level){
    unsigned int DAC_Word = 0;

    // Write to DAC, Gain = 2, /SHDN = 1, 12-bit value in low 12 bits
    DAC_Word = (0x1000) | (value & 0x0FFF);

    // drive /CS low on DAC
    P4->OUT &= ~BIT1;

    // Shift upper byte of DAC_Word 8-bits to right
    EUSCI_B0->TXBUF = (unsigned char) (DAC_Word >> 8);

    // is the USCI_A0 TX buffer ready?
    while (!(EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG));

    // Transmit lower byte to DAC
    EUSCI_B0->TXBUF = (unsigned char) (DAC_Word & 0x00FF);

    // Poll the TX flag to wait for completion
    while (!(EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG));

    // drive /CS high on DAC
    P4->OUT |= BIT1;

    return;
}
```

LCD_Test.c

```
//****************************************************************************
// Test proper connection and operation of LCD.
//****************************************************************************

#include "msp.h"
#include "LCD.h"
#include "timing.h"

// displays a msg of length n on the LCD
void displayMessage(char* msg, int length, int freq) {
    int i;
    for (i = 0; i < length; i++) {
        LCD_data(msg[i], freq);
    }
}

int main(void) {
    LCD_init(FREQ_24_MHZ);
    LCD_command(1, FREQ_24_MHZ); /*clear display */
    delayMs(500, FREQ_24_MHZ);
    /*set cursor at beginning of first line */
    LCD_command(0x80, FREQ_24_MHZ);
    displayMessage("Testing", 7, FREQ_24_MHZ);
    delayMs(500, FREQ_24_MHZ);
}
```

save_samples.c

```
/*Save samples and patterns to EEPROM*/
#include "msp.h"
#include "mytiming.h"
#include "stdint.h"

#define EEPROM_ADDRESS 0x50   //address of serial EEPROM

void InitEEPROM(uint8_t DeviceAddress);
void WriteEEPROM(uint16_t MemAddress, uint8_t MemByte);
uint8_t ReadEEPROM(uint16_t MemAddress);
uint16_t TransmitFlag = 0;

void main(void)
{
    uint32_t i;
    uint8_t value;

    WDTCTL = WDTPW | WDTHOLD;                  //Stop watchdog timer

    _enable_irq();                             //Enable global interrupt

    InitEEPROM(EEPROM_ADDRESS);

    WriteEEPROM(0x169E, 0xE0);  //address, content
    for (i = 4000; i > 0; i--); //Delay for EEPROM write cycle (5 ms)
    WriteEEPROM(0x169F, 0xD4);  //address, content
    for (i = 4000; i > 0; i--); //Delay for EEPROM write cycle (5 ms)
    WriteEEPROM(0x1700, 0xDA);  //address, content
    for (i = 4000; i > 0; i--); //Delay for EEPROM write cycle (5 ms)
    WriteEEPROM(0x1701, 0xD2);  //address, content
    for (i = 4000; i > 0; i--); //Delay for EEPROM write cycle (5 ms)
    WriteEEPROM(0x1702, 0xDC);  //address, content
    for (i = 4000; i > 0; i--); //Delay for EEPROM write cycle (5 ms)
    WriteEEPROM(0x1703, 0xD9);  //address, content
    for (i = 4000; i > 0; i--); //Delay for EEPROM write cycle (5 ms)
    WriteEEPROM(0x1704, 0xC9);  //address, content
    for (i = 4000; i > 0; i--); //Delay for EEPROM write cycle (5 ms)
    WriteEEPROM(0x1705, 0xE4);  //address, content
    for (i = 4000; i > 0; i--); //Delay for EEPROM write cycle (5 ms)
    WriteEEPROM(0x1706, 0xC2);  //address, content
    for (i = 4000; i > 0; i--); //Delay for EEPROM write cycle (5 ms)
    __sleep();                     //Go to lower power mode
```

```
}

// Initialize I2C bus for communicating with EEPROM.
void InitEEPROM(uint8_t DeviceAddress){

   P1->SEL0 |= BIT6 | BIT7;                //Set I2C pins of eUSCI_B0

   //Enable eUSCIB0 interrupt in NVIC module
   NVIC->ISER[0] = 1 << ((EUSCIB0_IRQn) & 31);

   //Configure USCI_B0 for I2C mode
   EUSCI_B0->CTLW0 |= EUSCI_A_CTLW0_SWRST;   //Software reset enabled
   EUSCI_B0->CTLW0 = EUSCI_A_CTLW0_SWRST |   //Remain eUSCI in reset mode
           EUSCI_B_CTLW0_MODE_3 |            //I2C mode
           EUSCI_B_CTLW0_MST |               //Master mode
           EUSCI_B_CTLW0_SYNC |              //Sync mode
           EUSCI_B_CTLW0_SSEL__SMCLK;        //SMCLK

   EUSCI_B0->BRW = 30;                       //baudrate = SMCLK / 30 = 100kHz
   EUSCI_B0->I2CSA = DeviceAddress;          //Serial EEPROM address
   EUSCI_B0->CTLW0 &= ~EUSCI_A_CTLW0_SWRST;  //Release eUSCI from reset

   EUSCI_B0->IE |= EUSCI_A_IE_RXIE |         //Enable receive interrupt
                   EUSCI_A_IE_TXIE;
}

//  Function that writes a single byte to the EEPROM.
void WriteEEPROM(uint16_t MemAddress, uint8_t MemByte){

   uint8_t HiAddress;
   uint8_t LoAddress;

   HiAddress = MemAddress >> 8;
   LoAddress = MemAddress & 0xFF;

   EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TR;          //Set write mode
   EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;       //I2C start condition

   while (!TransmitFlag);               //Wait for EEPROM address to transmit
   TransmitFlag = 0;

   EUSCI_B0 -> TXBUF = HiAddress;   //Send high byte of the memory address

   while (!TransmitFlag);           //Wait for the transmit to complete
   TransmitFlag = 0;

   EUSCI_B0 -> TXBUF = LoAddress;   //Send high byte of the memory address

   while (!TransmitFlag);           //Wait for the transmit to complete
   TransmitFlag = 0;

   EUSCI_B0 -> TXBUF = MemByte;     //Send the byte to store in EEPROM
   while (!TransmitFlag);           //Wait for the transmit to complete
   TransmitFlag = 0;

   EUSCI_B0 -> CTLW0 |= EUSCI_B_CTLW0_TXSTP;   //I2C stop condition
}

// I2C Interrupt Service Routine
void EUSCIB0_IRQHandler(void){

   if (EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG0){      //Check if transmit complete

         EUSCI_B0->IFG &= ~ EUSCI_B_IFG_TXIFG0;  //Clear interrupt flag
         TransmitFlag = 1;                       //Set global flag
   }

   if (EUSCI_B0->IFG & EUSCI_B_IFG_RXIFG0){      //Check if receive complete

         EUSCI_B0->IFG &= ~ EUSCI_B_IFG_RXIFG0;  //Clear interrupt flag
         TransmitFlag = 1;                       //Set global flag
    }
}
```

<u>output_selection.c</u>

```
//***************************************************************************
// Use a button to select between MIDI out (UART) or 3.5 mm out (DAC)
//***************************************************************************

#include "msp.h"
#include "mytiming.h"
#include "DriveDAC.h"

volatile unsigned int  button_flag_2 = 0;
volatile int saved_IFG_2;
volatile unsigned int TempDAC_Value = 0; //give this to DriveDAC for output

void setFreq(int freq);
void port_2_setup(unsigned char pinmask);
void UART0_init(void);
void SPI_init(void);
void Drive_DAC(unsigned int output)

void main(void)
{

    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer

    // Initialize Port 2 (button connections)
    port_2_setup(BIT4);  //MIDI output
    port_2_setup(BIT7);  //3.5mm output

    // Enable global interrupt
    __enable_irq();

    // Enable ADC and Timer A0 interrupts in NVIC module
    NVIC_EnableIRQ(PORT2_IRQn);

    while (1){
       if (button_flag_2){                    //button pushed, pick an output
          button_flag_2 = 0;                  //Reset flag
          P3->IFG = 0;                        //clear any pending flags
          if(saved_IFG_2 & BIT4){             //MIDI output selected
             UART0_init();
             while(!(EUSCI_A0->IFG & 0x02)) { }
             EUSCI_A0->TXBUF = uart_3;
             while(!(EUSCI_A0->IFG & 0x02)) { }
             EUSCI_A0->TXBUF = uart_2;
             while(!(EUSCI_A0->IFG & 0x02)) { }
             EUSCI_A0->TXBUF = uart_1;
             while(!(EUSCI_A0->IFG & 0x02)) { }
             EUSCI_A0->TXBUF = uart_0
          }else if(saved_IFG_2 & BIT7){        //3.5 mm output selected
             SPI_init();
             Drive_DAC(TempDAC);
          }
          //re-enable button interrupts for debounce
          P2->IE |= BIT4;                    //re-enable interrupts
          P2->IE |= BIT7;
       }
    }
}

void PORT2_IRQHandler(void)       //Interrupt handler for Port 2
{
       saved_IFG_2 = P2->IFG;
       button_flag_2 = 1;         //Set flag to signal button press detected
       P2->IFG = 0;               //Clear pending interrupt flag
       P2->IE  &= ~BIT4;          //Disable interrupt for debouncing
       P2->IE  &= ~BIT7;          //Disable interrupt for debouncing
}

void port_2_setup(unsigned char pinmask)
{
    P2->SEL0 &= ~pinmask;    // GPIO, not alternate function
    P2->SEL1 &= ~pinmask;
    P2->DIR  &= ~pinmask;    // input
    P2->OUT  |=  pinmask;    // pull up, not down
    P2->REN  |=  pinmask;    // enable pullup
    P2->IES  |=  pinmask;    // high->low transition
    P2->IFG  &= ~pinmask;    // clear possible stale IFG
    P2->IE   |=  pinmask;    // enable IFG
    return;
}

void UART0_init(void)
{
    EUSCI_A0->CTLW0 |= 1;       // put in reset mode for config
    EUSCI_A0->MCTLW = 0;        // disable oversampling
    EUSCI_A0->CTLW0 = 0x0081;   // 1 stop bit, no parity,SMCLK,byte
                                // data
    EUSCI_A0->BRW = 26;         // 3,000,000 / 115200 = 26
    P1->SEL0 |= 0x0C;            // P1.3, P1.2 for UART
    P1->SEL1 &= ~0x0C;
    EUSCI_A0->CTLW0 &= ~1;       // take UART out of reset mode
}
```