Using a Microcontroller to Create a Drum Machine

NJ Governor’s School of Engineering & Technology
Rutgers University

Robert Kertz
kertz.rob@gmail.com

Sarah Quinn
squinn27@ymail.com

Gareth Rose
g.rose6@gmail.com

Kevin Strotz
kevin.j.strotz@gmail.com

Penny Sun
penny.sun.94@gmail.com

Abstract

In the modern electronic instrument market, drum machines are often far too expensive and complex for operation by the everyday user. This project addressed this shortcoming by designing and building a user-friendly and cost-effective drum machine with an Atmel ATmega2560 microcontroller running the Arduino platform, a Voice Shield chip, and a limited amount of additional components. The device was coded using the Arduino variant of the C programming language in conjunction with additional platform-specific libraries. Extensive research was conducted throughout the project to better understand the hardware and programming processes and to optimize the final product in terms of price, simplicity, and functionality. The operational code for the machine was developed in function-specific portions, which were subsequently combined to form a single master code. Although complications were encountered during the development of the code, including those associated with array functionality, playback speed, sound storage, and memory access, all were resolved and the final prototype was fully functional. It was also established that in further product development, the most significant features that could be added are volume controls, greater flexibility in audio file importation, and the use of higher quality sounds.

1 Introduction

1.1 Microcontroller

Microcontrollers are vital to all modern day technological operations. They form the core of the integrated circuits and embedded systems that control everything from refrigerators to automobiles by operating the intricate electrical systems that exist in many modern consumer products. Microcontrollers contain a basic processor, flash (Random Access Memory (RAM)) and EEPROM (Electrically Erasable Programmable Read-Only Memory) memory, and both input and output peripherals that allow for interaction with the embedded system and any external components (see Figure 1). Microcontrollers are not to be confused with microprocessors, which contain full processing capabilities much like a Central Processing Unit (CPU) and are not designed for embedded applications. Microcontrollers are used because of they are compact and are often the most economical form of controlling a simple system. They take the place of separate processing, input/output, and memory units.  

1
1.2 Drum Machine

Drum machines have a similarly ubiquitous presence in the modern world, as they are used to produce much of the “pop” and “hip-hop” music that society enjoys today. A drum machine is an interface that allows an individual to create a drum beat containing multiple instruments by selecting beats in a sequence (see figure 2). This sequence typically represents one measure of music that is broken down into sixteen subdivisions acting as 16\textsuperscript{th} notes.

Unfortunately, these devices are often only available to professional musicians because they are unreasonably priced for regular consumers. These virtual drum machines generally require powerful computers with advanced processing ability and costly audio processing software. Consequently, such products are quite expensive and inaccessible to independent musicians. As virtual drum machines are replacing older, physical drum machines, this trend is becoming more pronounced. Additionally, physical drum machines are often limited to pre-loaded sounds and a set number of instrument tracks that can be run in the same beat.

1.3 Project Goals

To create a drum machine, using a microcontroller in conjunction with other peripherals, that:

- is accessible to independent musicians and standard consumers in that it is reasonably priced, user-friendly, and simple.

- is customizable through the ability to add new instruments and ancillary tracks.

- inspires creativity by offering functions such as saving beats and easily clearing the workspace.

- is compact and mobile.

2 Background

2.1 Interacting with Hardware

2.1.1 Arduino Mega 2560 Microcontroller

Microcontrollers are small, embedded circuits equipped with a processor. Able to read and execute code designated by the user, they offer unmatched potential for flexible and customizable programming, and do not have a
The specific microcontroller used in the development of this project was the Atmel ATmega2560 mounted on an Arduino Mega 2560 chip and running the Arduino platform. The Mega 2560 chip has 54 digital pins, including 14 pulse width modulation (PWM) pins, 16 analog input pins, five 3.3 volt power supply pins for external peripheral power, and several ground pins. It also contains four hardware serial ports, and a 16 megahertz microprocessor. Combined, these features allow the controller to receive external input, process internal commands in response, and generate output signals. Arduino microcontrollers can also interact with peripheral devices called “shields” that are designed specifically to enhance their functionality.

Throughout project development, the pins of the controller were used in different ways to facilitate different controller functions. The digital pins on the Arduino can be defined as either input or output sources by the user depending on how they are initialized in the program.

When not equipped with PWM, pins can read or output values of either zero or five volts, making them ideal to register input from buttons, which output simple on/off states. The digital pins equipped with PWM offer all the usage of the standard digital pins but can output any amount of power up to 5 volts as defined by the program. The analog pins are input-only and read values between 0 and 5 volts, allowing for a “fading” mechanism required when potentiometers are used as input devices.

### 2.1.2 LCD Screen

A liquid crystal display (LCD), is a commonly used type of visual display that manipulates layers of liquid crystals (LCs) to create different characters (see figure 3). The LCs themselves do not emit light, but rather block and alter the light produced by a backlight, which is usually an LED or a cold cathode fluorescent lamp. LCDs have widely replaced the use of Cathode Ray Tube displays, as LCDs are far cheaper, more compact, and more reliable. For the purposes of this project, a general alphanumerical 4x20 character, LED-backlit LCD screen was used. The screen itself is controlled by 16 pins (with the exception of 4 pins not being used for this project) connected directly to an output pin on the microcontroller (see section 3.1.3 for more details on LCD screen functionality). A specific C library (see section 2.2.2.1) was used so that certain cells on the screen could be manipulated by the code that supports screen navigation and graphics.

### 2.1.3 Voice Shield

The Voice Shield (see Figure 4) is a plug-in analog audio processing peripheral...
that allows users to store and play audio files when using the Arduino platform. The shield is inserted directly into the pins on the top of the Arduino microcontroller. However, unlike microcontroller systems, the Voice Shield contains solely memory storage and audio output units, rather than an actual processor. By creating the Voice Shield’s recording and playback functionalities in year, SpikenzieLabs built upon the existing Winbond ISD4003 ChipCorder voice chip technology. “Voice chip” technology is named for its ability to store human voices. One of its primary uses is in devices such as answering machines that provide a humanoid audio interface. SpikenzieLabs’s Voice Shield has helped to advance similar technology by introducing a more compact device that can hold up to four minutes of audio memory at a sampling rate of eight kilohertz. SpikenzieLabs’s Voice Shield has helped to advance similar technology by introducing a more compact device that can hold up to four minutes of audio memory at a sampling rate of eight kilohertz. One of its primary uses is in devices such as answering machines that provide a humanoid audio interface. SpikenzieLabs’s Voice Shield has helped to advance similar technology by introducing a more compact device that can hold up to four minutes of audio memory at a sampling rate of eight kilohertz.

Figure 4: A Voice Shield Board
Source: makershed.com

The Voice Shield outputs sound through a 3.5mm audio “out” stereo jack, enabling users to connect their own audio amplification devices. It also inputs sound through a separate audio “in” stereo jack, to facilitate audio file importation. Users can break up the Voice Shield’s four minute memory space into as many as 1200 unique addresses within its memory, although the default size is 80 sound slots. To store the sounds, the Voice Shield automatically inserts pointers into its memory to mark each audio file, decreasing the storage and processing demands on the Arduino chip itself. Furthermore, with this pre-programmed function, the Voice Shield compensates for differing file lengths by recognizing the length of recorded audio files and communicating this information to the Arduino.

2.2 Interacting with Software

2.2.1 C Programming with Arduino

The C programming language operates through an imperative style of computer programming; in other words, it allows the user to issue commands to an output device in a very controlled sequence. As a result, the C programming language enables the user to exercise greater control over how the memory is distributed. For the novice programmer, the language offers the most control and efficiency in a relatively simple format. Before the program can be sent to the output device, it must be compiled using the Arduino compilation code, which translates the C language so that it can communicate with the designated device.

C programming utilizes various forms of code syntax to execute different commands, such as variables, arrays, functions, and flow control structures. A variable in computer programming is similar to a variable used in mathematics in that it stores information. However, variables in computer programming are able to store more than simply numerical values. To provide the user with flexibility in writing code, computer programming variables receive a name, data type, and sometimes an initial value that together make each variable initialized unique. Data types specify the
way that a value should be stored to the memory and include bytes, integers, floats (decimals), characters, and strings of characters. In addition, variables can be defined “globally” at the beginning of the code, which makes them accessible throughout the entire program, or “locally” within a function, which restricts them to a certain segment of the code. An array is a type of variable that serves as an indexed collection of values. Arrays are useful in situations that involve one variable that is expected to have several different states or values throughout the execution of the program. Arrays are typically used within “for-loops” in order to cycle through a series of values.

A function is a block of code containing commands that together execute a complete action. Functions are used as a tool to reduce the length of the written code and therefore are often created when a process needs to be performed at multiple points in the code. When called by a trigger in the main code, a function will execute the block of statements it contains. These statements can return data values, operate under set parameters, process inputted information, and execute external commands.

Control statements are other mechanisms used to help to regulate the flow of program components. The simplest form of a control statement is the “if-statement”. An if-statement requires certain conditions to be met to in order to initialize a set of user-defined actions. If these conditions are not met, the statement will not execute.

A variation of the if-statement is the “if...else-statement”. An if-statement can only test one condition and completes one set of actions if the condition is true; an if...else-statement can test multiple mutually exclusive conditions, setting a default group of actions to perform if the initial if-statement’s condition is not met. If one condition within the if...else-statement is met, however, the program stops testing the other conditions. Consequently, if...else-statements allow for greater control than the basic if-statement because it tests for more behaviors and contains a default behavior as well.

A “switch...case-statement” is very similar to an if...else-statement in that it guides the program through a set of commands if a given condition is met and also can contain several mutually exclusive conditions. However, the main difference between the two is that while if- and if...else-statements can evaluate both conditions that span a range of values and conditions that compare only discrete values, switch...case-statements can only evaluate discrete conditions. Also, switch...case-statements do not automatically skip over the rest of the tests if one test has already been proven true, so a break (a command that tells the program to exit the switch) must be manually inserted at the end of each test.

The “if-”, “if...else-”, and “switch...case-” statements all control the flow of the code by executing specific commands when certain conditions are met. Loops such as “for-” and “while-” loops are employed for different reasons.

A for-loop, when initialized, will repeat a block of code for a user-specified number of iterations. In order to run, the for-loop requires three pieces of information: a starting value for a counter variable, a termination condition, and an increment expression. The first time the program
cycles through the for-loop, it sets the counter variable to its specified initial value. Then, the loop executes the entire block of code and returns to the for-statement, testing whether or not the termination condition is true. If it is true, the for-loop will terminate. If it is false, the loop will use the increment expression to change the value of the counter variable and then will follow the block of statements again. A for-loop is most useful in situations requiring repetitive actions.

While-loops are initialized by a condition and continuously cycle through a block of code until that condition is no longer met. They can also be maintained until a trigger indicates to stop the function or state. For instance, when using this loop, pressing a button can begin a process that is continued until the button is pressed again. The default loop function essential to the Arduino platform (explained below) is a while loop that always runs because it generally does not have specific triggers to register.

Figure 5: Sound_test2 Arduino Sketch demonstrating the loop and setup functions, as well as the Voice Shield Library play functions

Within the Arduino system, a computer program is called a sketch. Each Arduino sketch contains two necessary components: a setup and a loop function (see Figure 5). The setup function runs only once each time the Arduino board powers up or resets, and is mainly used to prepare the initial state of the program by declaring pin statuses and variables. The loop function continuously runs the commands programmed into it, allowing the code to react, regulate, and adjust to peripheral input. The Arduino sketches are also compatible with various libraries, specialized collections of commands that are used to communicate with various pieces of hardware or to simplify code.

2.2.2 Library Functions

LCD Library: <LiquidCrystal.h>

The Arduino LCD library includes commands that control the processes of navigating around and printing to the LCD screen. Specific commands are used to make certain cells active for modification, for placing a blinking cursor in a cell, or for printing characters to cells.

Voice Shield Library: <VoiceShield.h>

The Voice Shield library (see Figure 5) is the collection of commands available to the Arduino that let it communicate with and control the Voice Shield. The library enables the user to divide the Voice Shield’s memory into a specific number of sound slots, record audio files to these slots, and play back these files in response to commands in the code.

EEPROM Library: <EEPROM.h>

The EEPROM library includes the commands necessary to write information to
EEPROM memory and retrieve the same information later. These commands are essential to the successful functioning of the memory. The functions cooperate to save information to and read from a specific address (a numerical value) in the EEPROM\textsuperscript{11}.

2.3 Interacting with controller memory

2.3.1 EEPROM

Electrically erasable programmable read-only memory (EEPROM) is a medium in which information can be saved and recalled using the EEPROM library code\textsuperscript{12}. This EEPROM library was used frequently throughout the project and incorporated into the drum machine’s programming code so as to save user-created beat patterns. Unlike flash memory, EEPROM retains the information saved to it even when power is cut from the memory unit. This allows users to save beats that can later be recalled and re-opened\textsuperscript{13}.

Information must be written to EEPROM one byte at a time\textsuperscript{14}. A byte, one of several data types, is a collection of 8 bits, and each bit has a value of 0 or 1. In the byte, every bit has a numerical value based on powers of 2; from right to left, the first term has a value of $2^0$, or 1, the second term has a value of $2^1$, or 2, etc. When the bit value is 1, it is considered “on”; when the bit value is 0, it is “off”. In a byte, all of the “on” bits are added together to assign it a value anywhere from 0 to 255\textsuperscript{15}. The Arduino Mega 2560 used to control the prototype is limited to a maximum memory capacity of 512 bytes.

One drawback of EEPROM is that it does not operate perpetually as it can only endure 100,000 read/write cycles before it fails to function\textsuperscript{16}. If a user designs a program that needs to save over a hundred bytes at a given moment, the 100,000 cycle limit approaches more quickly than might be otherwise expected.

2.3.2 Flash

Flash memory is another essential form of information storage. One of flash memory’s positive characteristics is its speed. Flash memory writes and reads to and from memory much faster than EEPROM because it is able to save information in large blocks, whereas EEPROM must save information byte by byte\textsuperscript{17}. However, unlike EEPROM, flash memory cannot retain information when power is cut from the memory unit.

For the purposes of this project, the beat arrays created in the user interface are automatically saved to the flash memory. Because of this, if the user decides diverts from the workspace to the menu and returns to the interface afterwards, the beat pattern will reappear on the screen as if untouched. Flash memory will hold these beat values within the three arrays as long as the system is not powered off.

3 Method

3.1 Major Development Sections

3.1.1 Design and Concept Development

Developing a simple and functional user control interface was vital to achieving the project’s main goals since peripheral selection has the most drastic effect on project cost, operating simplicity, and system efficiency (see Figure 6). Unlike traditional drum machines that use several sets of buttons and individual lights to signify user selections, it was decided that
greater efficiency, affordability and flexibility could be achieved if an LCD screen was used to represent user selections (see Figure 7).

The prototype was designed so that users would spend most of their time in the main workspace, the space that oversees beat modification. In this area the user can adjust when specific instruments are played and how many beats they are played for. The operator navigates between the instrument tracks, which are oriented horizontally, and the 16 sequential beats on which he can place notes (see Figure 8). Selected beats are denoted by an asterisk on the display. Furthermore, to provide customization opportunities, the display can be switched to an interactive menu (see Figure 7), from which work can be saved, previously saved files can be opened or cleared, and a new workspace can be initiated.

Because many of the user actions required to create beat patterns are repetitive or complementary, only a few simple buttons are needed to make up a comprehensive user interface. Four directional buttons control the position of the cursor on the screen; the select and deselect buttons allow users to edit notes in the workspace and enter menu functions; a menu button enters and exits the menu display; a play button plays the beat sequence; and a stop button halts that playback. Additionally, a separate potentiometer is used to control the speed of the beat.
Making the interface operational included two distinct phases: wiring the external hardware to the power and peripheral pins on the controller, and writing the code that recognizes the hardware inputs and carries out resulting operations. The only input peripherals used in this project were buttons and potentiometers. Buttons open or close circuits in order to allow current to flow (see Figure 9). Potentiometers allow a constant current to flow but vary the amount of resistance applied to the signal (see Figure 10).

To begin the hardware wiring stage, each button was wired to a power source, ground, and signal output directed to one of the digital input pins on the microcontroller. In the setup section of the code, the digital pin for each button was designated as an input to enable the microcontroller to receive information from that peripheral. Global variables were defined afterward to allow the program to register a signal from a button and trigger appropriate commands (see Figure 11). To lessen programming confusion, most global variables were named after their function. The potentiometer was similarly wired, with a power source, signal output, and ground (see Figure 12). Unlike the buttons, however, the signal output was wired to an analog input rather than a digital one.

```c
int assayThree[19];
int assay[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int playButtonPressed=0;
int playButtonState=0;
int resetButtonPressed=0;
int BOB;
int KnobReadings;
int btnUPpin = 35;  //changed
int btnDOWNpin = 34;  //changed
int btnLEFTpin = 33;  //changed
int btnRIGHTpin = 32;  //changed
int btnSELECTpin = 31;  //changed
int btnDIRLPin = 30;  //changed
int btnDIRRPin = 29;  //changed
int btnFLYPin = 28;  //changed
int btnREMPin = 27;  //changed
int screenMate;
int xps;
int yps;
int x;  // used in reload code

void setup(){
    lcd.begin(20, 4);
}
```

![Figure 11: Example of Global Variables](image)

After all of the hardware was wired and defined as input sources, the corresponding functions to the recognition of button and potentiometer signals were written. Each directional button was programmed to move...
a cursor around the screen by altering “xpos” and “ypos,” the x and y-axis positions of the cursor (see Figure 13). The play button was programmed to start audio playback and set the cursor in motion along the bottom of the screen as a method of tracking beat progression. Likewise, the stop button was programmed to terminate the execution of the play command, able to end the emission of sounds at any point (see 3.1.4 Sequencing for more information on the play and stop commands). Additionally, the menu button was programmed replace the workspace with the menu screen when pressed once and to return to the workspace if pressed again (see 3.1.3 Graphics).

```c
while(screenstate==1)
{
    x=0;
    lcd.setCursor(1, 0);
    lcd.print("H");
    lcd.setCursor(1, 1);
    lcd.print("S");
    lcd.setCursor(1, 2);
    lcd.print("K");
    lcd.setCursor(2, 3);
    lcd.print("1234567890123456");
    lcd.setCursor(xpos, ypos);
    lcd.setCursor(1, 0);
    if(digitalRead(buttonPin)== HIGH)
    {
        lcd.setCursor(xpos, ypos+1);
        lcd.setCursor(1, 0);
        ypos = ypos+1;
    }
}
```

Figure 13: An example of setting up the default graphic interface (lcd.print) and the code for cursor movement (lcd.setCursor)

The most intricate programming involved the “select button,” which has a variable function depending on the position of the cursor and the screen state (menu or workspace). The select button, if pressed while the cursor is positioned over a cell in the workspace, will place an asterisk there to be read as a sound during playback. Yet if a menu is active, pressing the select button opens another menu or enables the menu function that the cursor is adjacent to. The deselect button, which is only active in the workspace, removes notes that were previously selected. Finally, the playback speed control knob was programmed to affect the delay that determines playback speed (see section 3.1.4 for more information about sequencing and playback and section 3.1.5 for more information about the playback speed control function.)

3.1.3 Graphics

Because the screen is the only output device that allows the user to visualize their beat sequence and the various menu options, designing a simple and effective graphic layout was very important to the functionality of the project. The main screen setting, the workspace, was designed to make it easy for a user to navigate around the screen and place notes in different beat sections and instrument tracks. Each of the three instruments is paired with a horizontal track that is divided into 16 columns. These columns represent the individual 16th-note division of the full measure that is played continuously by the drum machine. For identification purposes, each row is preceded by the first letter of the instrument in that track, and each 16th-note column is numbered at the bottom of the screen. Notes selected within each column are represented by asterisks (see Figure 14).

Figure 14: Workspace with asterisks designating beats for each track. H, S, and K, represent the instruments, whereas 1-16 number the notes
Perhaps the most compelling reason to use an LCD screen instead of a more complex button interface with LED light indicators, is the ability to include menu functionality. As was previously mentioned, pressing the “menu” button in the workspace calls a main menu that includes four choices: Save, Open, Clear, and New. Selecting “Save” changes the display to the save menu, which allows users to save their beat sequence to one of four EEPROM memory slots. On the other hand, selecting the “Open” menu allows users to open previously saved beats, while choosing “Clear” enables users to clear such beats. The “New” function clears the current workspace, making the drum machine more efficient as the user can clear without solely using the deselect button.

In order for the screen to become operational, the LCD was wired into specified pins on the controller (see Figure 15) that were defined as LCD outputs within the setup code (see Figure 16). Code was then written to print the default characters on the workspace, the cursor, the changing menus, and the asterisks on selected beats.

```c
int x; // used in reload code
void setup() {
  lcd.begin(20, 4);
  pinMode(btnLEFTpin, INPUT);
  pinMode(btnRIGHTpin, INPUT);
  pinMode(btnSELpin, INPUT);
  pinMode(btnUNDPin, INPUT);
  pinMode(btnPLAYpin, INPUT);
  pinMode(btnRESETpin, INPUT);
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(10, OUTPUT);
  pinMode(A10, INPUT);
}
```

**Figure 16:** Example of setup code to define peripherals as inputs or outputs

### 3.1.4 Sequencing

Designing the program sections that govern the unseen aspects of the drum machine played a vital role in establishing its functionality. These programming processes focused on the play and stop functions as well as the use of flash memory to keep track of note selections.

In order to select beats, on and off values have to be saved for every cell. A beat that is selected is designated as “on” and a beat that is not selected is designated as “off”. Information about the location of selected beats is stored into flash memory using three arrays. Each instrument corresponds to a separate array with 16 slots, and within each array, a 1 represents a selected “on” note and a 0 represents a silent “off” note. The play function accesses these individual values within the arrays to decide which notes to play.

The process behind saving to flash memory is triggered by input from the select and deselect buttons. When the drum
machine is turned on by an external power source, the arrays hold all 0 or “off” values. However, if a cell in the workspace is selected afterward, an asterisk is placed, and the code inserts a 1 into the corresponding position in that instrument’s array. The deselect function replaces a 1 in an array with a 0 to turn a beat off. As a result, when the play button is pressed, an if...else-statement scans through the arrays for 1’s and directs the Voice Shield to play sound files when it recognizes a selected cell (see Figure 17).

![Figure 17: The playing code used to cycle through arrays and determine which beats are on and off. If a certain beat is on, a sound is played.](image)

Between each note, the program delays for a certain amount of time determined by the playback speed knob. As the delay time changes the rate at which the Voice Shield plays sounds becomes faster or slower. The play sequence is ended when the stop button is pressed and the Voice Shield executes a command to end sound emission.

3.1.5 Playback Speed

Because tempo is an incredibly important musical aspect of a beat sequence, user input of playback speed was recognized as an important factor in project development. A potentiometer was wired into an analog input on the controller, which would be able to read and transfer the signal determined by the rotation of the potentiometer. In order to allow the user to visualize speed modulation, several LED bulbs were placed in a line, and light up sequentially, increasing in intensity as the playback speed is increased. The rotation value read by the analog pin is interpreted by a conversion function (see Figure 18) that creates an appropriate delay value out of the input electrical section. The conversion function is needed because as the delay needs to decrease as the rotation value increases. Thus, this situation requires a negative correlation between analog read and delay, whereas, normally, a positive correlation is present.

```c
if (analogRead(A10) > 409 && analogRead(A10) < 614) {
  delay(1000);
  analogWrite(6, 255);
}
```

Figure 18: Speed of Play change function that assigns delay times to rotation values.

In order to affect the playback speed of the beat, this delay value is inserted in between notes in the play sequence. During playback, each note position has the possibility of either being a note or an empty space. If the position contains a note, the corresponding sound has to play and then wait for the sequencer to proceed to the next time position. Otherwise, the sequencer needs to simply wait for the set delay time.
before assessing the next column for beats. According to the combination of the play and playback speed control functions, the Arduino loops through each array of notes at the predetermined rate, playing appropriate sounds where notes are assigned and waiting in blank spaces.

3.1.6 EEPROM Memory

The incorporation of EEPROM into the drum machine was divided into two distinct parts: writing to EEPROM and reading from it. When a user wanted to permanently save a beat sequence, the three 16-slot arrays needed to be transferred byte by byte into the EEPROM, a task addressed by the addition of a “save” function. The function scans through each array one element at a time and saves each byte to memory slot. In the EEPROM memory units, each byte has a value of either 0 or 1 for each position in the arrays used by flash memory. As noted before, a 1 represents an “on” note, whereas a 0 denotes the absence of a note.

An “open” function was then created in order to read the bytes stored in the EEPROM and load them back into the arrays. As was the case with the write function, the EEPROM read function works byte by byte, meaning that each element is individually added into a position in its designated array. An additional function was included to analyze the arrays that had been read from the memory unit and to represent the selected positions graphically by re-inserting asterisks in the selected cells. Once the track arrays and the graphic note visualization in the workspace had been restored, a user could play and re-edit the loaded beat.

4 Analysis

4.1 Development Complications

4.1.1 Merging segment code

Throughout the research, testing, and early programming processes, several different sections of code were developed to address individual aspects of the total drum machine program. For instance, the graphics code that operated the LCD was originally isolated from the code for the sequencer. As a result, the separate sections of the code, including functions for playing sounds, saving and opening sounds, and controlling the LCD screen, had to be combined into one master code in to create a working program.

This compilation required not only the transfer of several codes into a single file, but also the alteration of variables and the establishment of new connections between code segments. For instance, while variables in the different pieces of code often referred to the same integer or aspect, they usually did not have the same name. Thus, all of the variables needed to be examined and renamed accordingly. The x position of the cursor, for example, was an integer important to more than one region of the code. It was found that in some parts of the code, x position was referred to as “xpos” and “cursorPositionX” in other parts. To introduce uniformity, “xpos” became the universal variable and “cursorPositionX” was removed.

Furthermore, functions within the code needed to be connected to each other through cause-and-effect relationships in the loop. Functions that had been created to sequence the sound needed to be triggered by an event occurring on the LCD screen or the pressing of a button. To accomplish this,
new global variables were introduced and, using the original LCD program as the skeleton code, all of the functions were placed into their proper context.

4.1.2 Interface functionality

When working with simple tactile buttons, a common issue is a hardware flaw called “contact bounce,” also referred to as “chatter.” This occurs over the course of the approximately 300 millisecond period it takes for a user to fully depress and release a button. The charge between the internal contacts (which are being brought together by the button depression to complete the circuit and produce an output signal) bounces back and forth several hundred times before the button is fully pressed and released. Consequently, this completes the circuit and produces an output signal several hundred times, registering as several hundred button presses on the microcontroller. In order to prevent chatter, complex “de-bounce” code is often run in order to tell the program to only accept one input signal from the button.

Because this code is often lengthy, time consuming to write, and prone to error, it was decided that a more simplistic approach to preventing chatter was necessary. Rather than running full de-bounce code, a 150 millisecond delay was added to the end of the operations loop that contains all of the button input commands, thus requiring the program to wait for 150 milliseconds after registering one button press before it begins to look for any other input.

4.1.3 Voice Shield functionality

The Voice Shield, which is connected to the Arduino microcontroller and a power source, has drum beats recorded onto it using the VS Programmer Lite, a piece of peripheral management software. Before opening the application, the VS loader program needs to be uploaded to the Arduino board so it can process the saving of audio files. Furthermore, all of the sound files have to be placed into a “Sound Bytes” folder and written into a “SoundScore” text file with the desired save slots designated. After this is accomplished, the sounds are uploaded through the VS Programmer. There were several major difficulties in programming the drum sounds although most were caused by slight typographical errors in the code.

After recording the sounds onto the Voice Shield, various complications were encountered. Three different drums were always loaded onto the device, including the kick drum, the snare drum, and either a tom-tom or a hi-hat. To test the functionality of the sequencer, beats were selected on the LCD, thus turning them “on” within the track arrays, and the play button was pressed. In most circumstances when the play function was initiated, only the kick and snare sounds would work, but not the hi-hat or tom-tom sounds in array one. At first, this seemed to be a problem with the array, but when the kick was used in array one, the sound played in accordance with the beats selected. Thus, it was determined that the code was not the issue, and that problems were stemming from either the Voice Shield hardware or the VS Programmer software.

In addressing this finding, it was thought that the issue might lie with the audio save slots on the Voice Shield. To combat this, the hi-hat track was moved to various other save slots in array one. Up to ten distinct sound slots were tried with no improved result.
Next, different sound files were substituted for the high-hat sound, with the thought that the delay within the sound files might be the cause of the problem. However, none of these sounds made an impact on the way that the drum machine was playing. It was eventually determined that the issue was directly related to the VS Programmer as the first sound programmed would always have an additional delay that would prevent it from being heard. As a result, a buffer sound was programmed first, followed by the three actual audio files, resolving the playback issue.

4.1.4 Playback Speed Control

Because tempo is such an integral component of music, the user needed to be able to control and visualize the playback speed. In the initial project plans, it was decided that a four digit digital display was to be used to display the speed in beats per minute (BPM). However, it was soon found that much of the internal circuitry hardwired into the only available four-digit display was either unreliable or not functional at all.

The approach was modified so that the user, by turning the input knob, could select one of ten preset speed settings for playback. For the display, a single digit digital display was to be used, which would show the digits zero through nine in correspondence with the selected rate of play. Unfortunately, this display also presented hardware issues that rendered it ineffective for our purposes.

Finally, it was decided that a line of small LED bulbs would be used and would light up sequentially and increase in intensity as the user increased the playback speed.

4.1.5 Flash Memory/Array Functionality

When the master code was first created, a problem surfaced: when a user would move away from the workspace by accessing a menu, the notes that they had previously selected would not reappear on the screen. Although at first this was thought to be a problem with the flash memory, it was determined that it was instead an issue with the arrays. The arrays were designed to save either a “1” or a “0” for each beat to turn it on and off. Thus, if a beat in the array was equal to “1”, the screen was supposed to display an asterisk at that beat. The asterisks would appear initially on the workspace, but after the user navigated away to the menu screen and then came back to the workspace, they would disappear. Some code was written to read the values of the arrays from flash memory and write them back into the arrays when the user returned to the interface from the menu.

However, even after this code was written, the asterisks did not reload onto the workspace. After the code was thoroughly examined, it was found that the arrays had been paired with incorrect “ypos” values, which was preventing them from being reloaded properly. This code issue was quickly resolved and the graphic reloading feature became functional.

4.1.6 EEPROM functionality

The functionality of the EEPROM saving system was challenged by programming limitations during the development of the machine. At first, the EEPROM did not seem to write the arrays of notes displayed on screen, for when the “save” function was called, the machine simply froze and had to be restarted. Additionally, the “open” function would cause the screen to blink excessively without
actually loading a saved drum pattern. After some investigation, it was discovered that, due to hardware limitations, arrays could not be written directly to memory and that each individual element of the array had to be individually defined and written to EEPROM. The code had to be altered so that it could save the 16 specific values of each array instead of saving the entire array itself.

4.2 Evaluation of project final state

Although complications were frequently encountered throughout the final coding process, they were all successfully confronted. Some problems, such as the array functionality issue, were easy to solve, while others, such as the difficulty with the VS Programmer, were challenging to identify.

4.3 Cost Benefit Evaluation

One of the major goals of this project was to create a commercially viable product for use by amateur musicians. To that end, a VoiceShield™ Slim Kit ($45.95), a 4x20 LCD Screen ($17.50), an Arduino Mega 2560 ($60.00), fourteen resistors ($3.40), nine buttons ($6.00), five LED lights ($4.20), two breadboards ($30.00), two knobs ($4.00), and approximately five feet of wire ($0.50) were used to build an affordable machine. The total cost was $171.55. Although the quality of the sound produced by the VoiceShield is not comparable to other existing drum machines, the simplicity and clarity of this design gives the drum machine an edge over existing products, which typically cost between $500 and $2000\(^1\).
implementation. One of the most intriguing of such ideas is the incorporation of additional instruments in the machine. By introducing a larger variety of sounds, the user would gain the opportunity to be much more creative in the construction of drum beats. Such sounds might include tom-toms, cymbals, different snare drums, and other types of bass drums. These sounds would require additional coding to allow the LCD screen to scroll up and down, as currently only three rows of the screen at a time can be used for instrument tracks.

Another significant improvement that could be made is the inclusion of higher quality audio components. The VoiceShield system that was provided has a limit of the sound quality it can retain, causing the drum sounds saved to it to become relatively distorted and low quality. With a larger budget, higher quality components could be used to produce more realistic drum sounds.

Yet another possibility would be the addition of a digital display of the BPM (beats per minute) at which the beat is playing. This would allow the user to easily recognize how fast a beat is playing. Again, a larger budget would allow the purchase of a fully functional three digit digital display that would adequately accomplish this task.

Furthermore, the drum machine itself could be much more visually appealing. Due to a lack of time and materials, we are left with a prototype. In this prototype, all of the drum machine’s wires are exposed. Provided more time and materials, the connections made by the wires could be embedded in a circuit board, creating a much more simple and clean machine. With more money, a case could be made to encase the electronics, ensuring the stability of the wires and creating a better looking drum machine.

6 Acknowledgements

The research team involved with this project could not have had such an enriching experience if it were not for the entire NJ Governor’s School of Engineering and Technology program. In particular, we would like to thank Mark Sproul and Alex Weiner, our Project Mentors, Sarah Masters, our GSET RTA advisor, Daniel Cobar, the GSET Head Counselor, Jean Patrick Antoine, the GSET Program Director, Dean Ilene Rosen, the Rutgers Associate Dean for the School of Engineering’s Office of Student Development, Rutgers University, the Rutgers University School of Engineering, Morgan Stanley, the State of New Jersey, Lockheed Martin, Automated Control Concepts Inc., Silver Line Building Products, Sharon Ma and Nan Yao (parents of Johnathan Yao, GSET 2010), the Tomasetta family, and Laura Overdeck for their help, encouragement, and continued support of the program.

7 References


4 "DNA TECHNOLOGY - Interfacing LCD to 8051." DNA TECHNOLOGY - Home.


