ST-080

MILESTONE E

Tinotenda Chemvura – CHMTIN004

Othniel Konan – KNNOTH001

Hermann Kouassi – KSSKOU001

John Odetokun – ODTJOH001

Team 6

EEE3074W – Embedded Systems

Table of Contents

Plagiarism Declaration

- We know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is our own.
- We have used the IEEE system for citation and referencing.
- In this report, all contributions to, and quotations from, the work(s) of other people have been cited and referenced.
- This report is our own work.
- We have not allowed, and will not allow, anyone to copy our work.

Signed  Dated 18 January 2017

Signed  Dated ……18 January 2017

Signed _____ Dated _____

Signed _____ Dated _____

Introduction

Background

Make great beats on the move; where you choose: A couple of the main advantages of a drum machine. The Drum Machine, in the context of this project, is an instrument that incorporates a drum set (Or multiple drum sets) and a recording machine to imitate the sounds of a drum kit when pads are struck. Drum Machines are often used by musicians, DJs or in general people who may not necessarily know how to play drums but are, however looking for a drum effect in the musical project they are busy with.

The aim of this project is to is conceptualize, design and implement a prototype drum machine that will demonstrate the full criteria of the EEE3074W design project.

Concept

A Drum Machine is an electronic musical instrument that is designed to imitate the sound of drums, cymbals and other percussion instruments. Drum machines are used in many different avenues of life. The use of a drum machine ranges from personal use to make nice sounding beats to professional commercial use by DJs.

Our prototype, the ST-080:

In its simplest form the ST-080 works by linking different pads/buttons to different percussion sounds (Digitized waveforms) and then using DACs and ADCs to convert the pad press to an amplified sound through speakers. The prototype being developed is a combination a digital drum set as well as a composer. This means that the user can compose beats and play them back to themselves. The layout of button pads and the drum machine as a whole allows for a simplistic user interface. This allows for excellent beats to be made with ease.

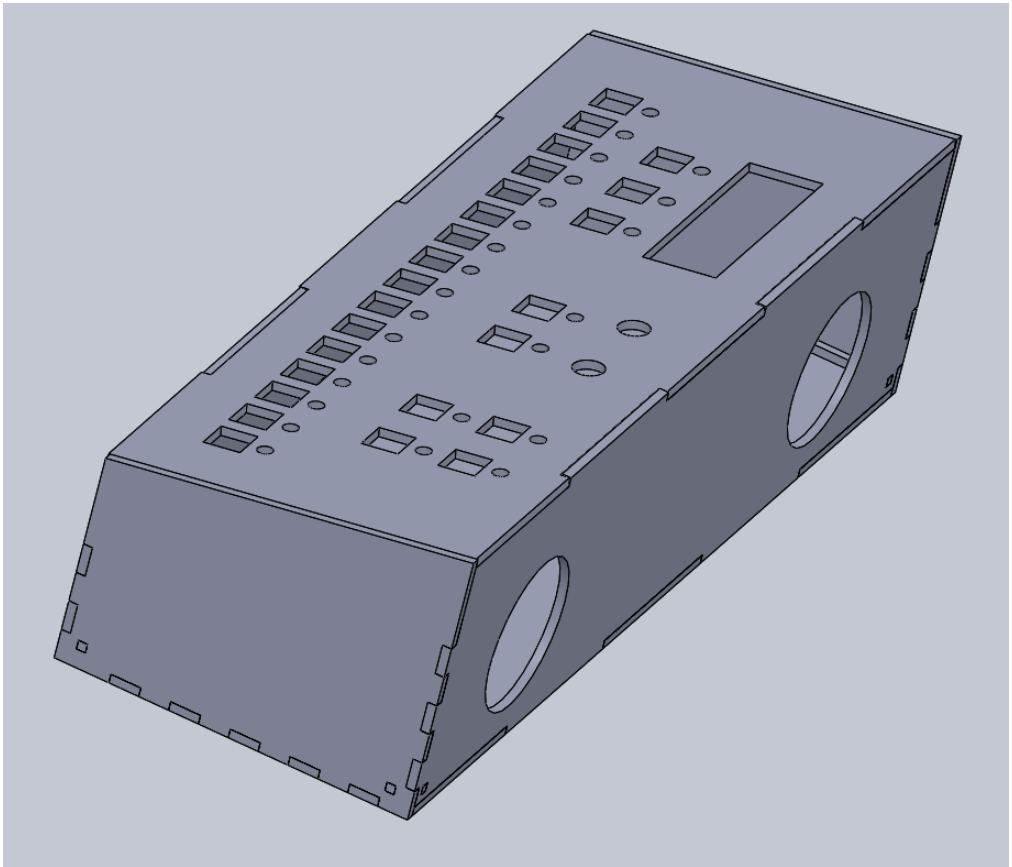

FIGURE 1 CONCEPT DESIGN

Design Overview

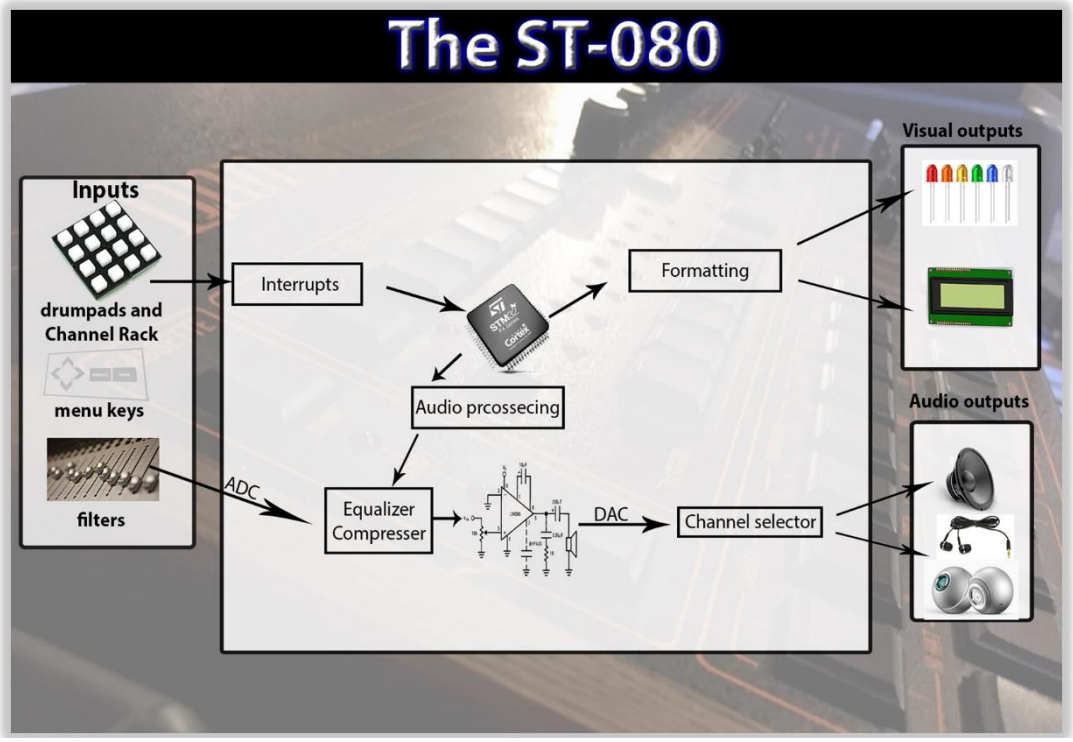

FIGURE 2 VIRTUAL POSTER SHOWING OVERVIEW OF SYSTEM

The overall design of the ST-080 is shown in **Figure 2** above which depicts the different sections and parts of the system. This including the inputs, processing and outputs. The design is based on the design of a very popular Rhythm Composer, the Roland TR-808, as seen in **Figure 3**. The TR-808 was one of the earliest programmable drum machines, with which users could create their own rhythms

FIGURE 3 TR-808 INSPIRATION FOR DESIGN

and beats.

The drum machine will incorporate four different instruments. Each instrument will have its own unique pre-recorded sample. The ST-080 will include a composer mode that allows for interactive beat creation. The user will be able to add an instrument to a channel rank while hearing the effects that it has on the beat. An example of this can be shown here [external link: https://www.youtube.com/watch?v=Cer1un9vciI]. This feature is useful for people who cannot play

the four instruments at the same time or for users that may not have the best drumming talent or rhythm. [1]

The drum machine will be given multiple modes in order to accommodate more features in a simple design.

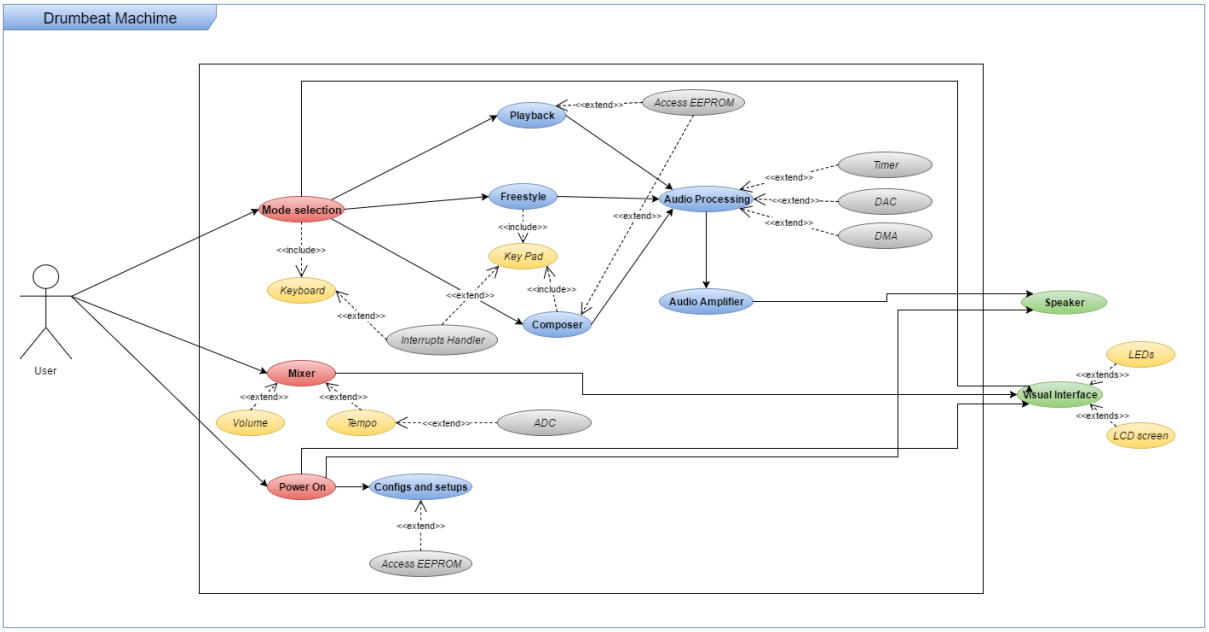

FIGURE 4 USE CASE DIAGRAM

The use case diagram, **Figure 4**, illustrates all functionality of the **Drumbeat Machine** as well as all physical interaction with the system.
The primary actor is the **user**.
The user will interface with system by mean of a **keyboard** and a **keypad**. Those two will respectively enable the user to select one of the various mode offer in the menu and play/record a beat or set of beats. The user's inputs will be captured by mean of GPIO pins and interrupts (Free-style mode) and any audio related selection will be processed by the audio processor. Appropriate UI messages will be displayed by the **LCD screen** and a **set of LEDs**. As for audio output, it will be amplified and outputted using a **speaker**.

Prototype Overview

The prototype of the ST-080 will be similar to (just as the name is) yet more simple than the TR-808. The user will have one of two options when using the ST-080, namely Freestyle mode and composer mode. The ST-080 will not have all the tuning capabilities of the TR-808. While there is room for filters and amplifiers to be added this product will not include such is it is just a prototype.

The prototype will consist of a Housing box that contains button pads, and LCD screen, LEDs and potentiometers. It will have a row of buttons at the bottom that will be used for the channel rack. These GPIOs will simply set a pin high or low which will indicate whether the instrument is selected to be played. In the top right corner there will be for buttons that will be used as interrupts in Freestyle mode. Whenever these buttons are pressed a sound will almost immediately be played.

The ST-080 prototype will have the following features:

- **Composer Mode**: This is a mode which allows the user to make use of the channel rack to compose an instrumental. They do this by pressing the buttons to create patterns (called

"scores") for the different instruments available on the device. The channel rack itself is designed for music with a time signature of 4 beats per bar hence it has 1 bar or 4 beats with each beat divided into 4 to make 16 pattern keys. The ST080 loops this array when in composer mode.

- **Playback Mode**: This mode allows the user to listen to any of the songs stored in the 16 slots provided on the ST080. Like the composer mode, the ST080 loops the channel rack when playing the music.
- **Freestyle Mode**: This mode allows the user to play around with the instruments on the ST080 and create their own music without any restrictions on tempo, patterns and time signature. They simply hit a button and the sound plays.
- **Permanent Storage**: The ST080 allows the user to store up to 16 songs. It does this by storing the musical scores that the user created rather than store the actual audio. This is more efficient because less storage space is needed and thus a cheaper storage device, like EEPROM, can be used to store music.
- **Varying Tempo**: The user can compose music at different tempos ranging from 40 up to 140 beats per second by simply turning a tempo-potentiometer on the ST080.
- **Headphones Jack**: The user has the option to plug in head phones or an external speaker system to either listen to the music in the headphones or connect to a larger sounds system. The output on the headphones jack has enough power to drive even the biggest studio headphones.
- **Feedback**: The LCD displays text to help the user with certain functions of the ST080. It also provides feedback like when it is saving something, loading, if no song is stored on a specific slot, the current mode, etc…). There are LEDs to show which keys have been pressed on the channel rack, which mode the user is in, which instrument is being modified, which song is playing or when it is saving or resetting.
- **Visual Metronome**: Instead of an audible metronome to help the user compose the music, a visual metronome was implemented instead. The LEDs on the channel rack will light up when that key is being played so the user can keep track of where on the channel rack the ST080 is playing music at and also where to add more scores as they wish.
- **Reset device**: There is a button to restart the device without saving any unsaved changes made to the songs.
- **Clear all songs**: There is a key combination that allows the user to delete all the songs stored in **RAM**. **NB**: the user would have to save if they want to remove these songs in the permanent storage too.

Many features are missing from the prototype due to time and budget constraints but there is space for all of them to be added if development of this device is to continue. Below is a picture of the ideal ST080 followed by pictures of the actual prototype that was built:

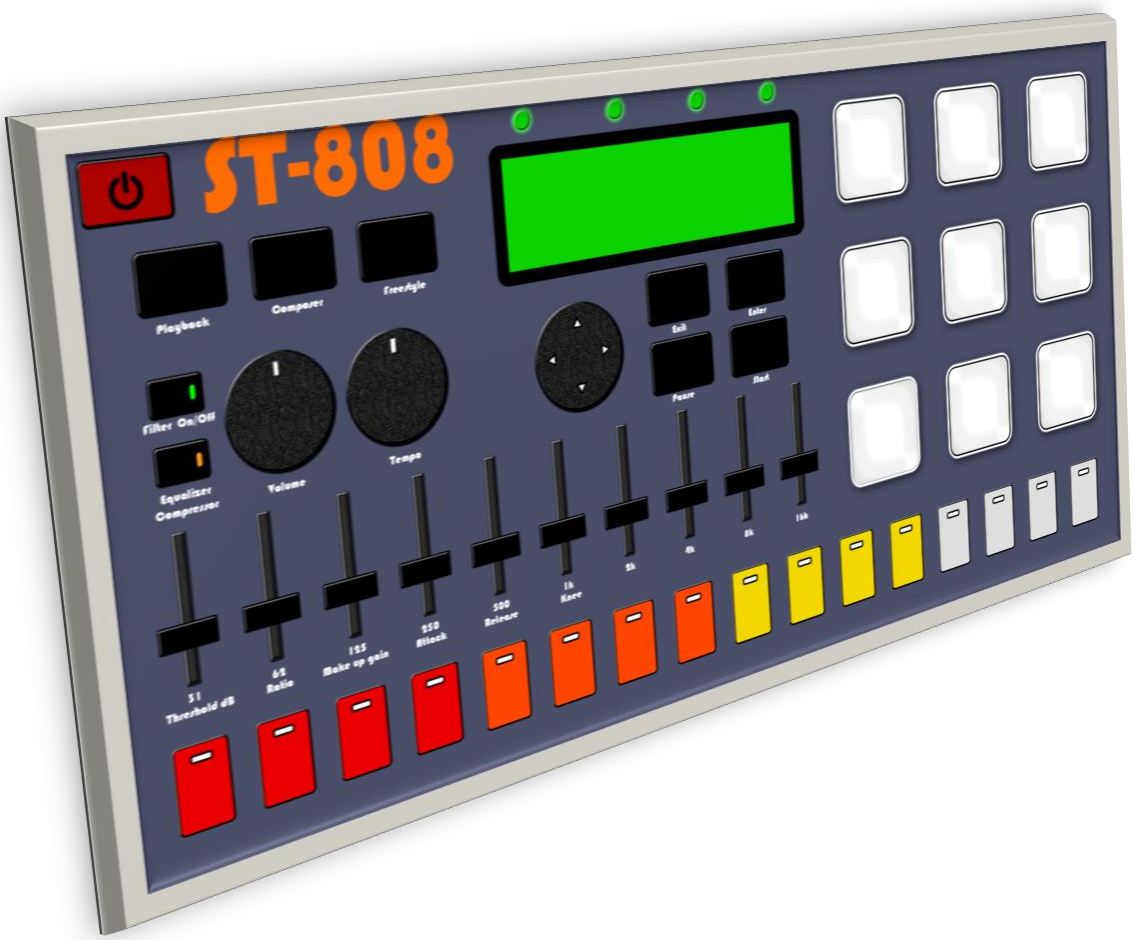

FIGURE 5: A COMPUTER GENERATED IMAGE OF THE IDEAL ST080

FIGURE 6 FINAL DRUM MACHINE

FIGURE 7 FINAL DRUM MACHINE, DIFFERENT ANGLES

TABLE 1: REQUIRED COMPONENTS

COMPONENT NAME	QUANTITY
STM32F4 discovery board	1
Pushbuttons	25
LEDs	25
LCD screen	1
Potentiometer	4
Speaker	2
64K EEPROM	1
LM386	1
Polarized capacitors	5
Capacitor	4
Resistor	53
Audio connector type d	1
Relay	1

Refer to Appendix for full Bill of Materials

Design

Hardware and Software Partition

Something that has been given plenty thought throughout the duration of this project has been the Hardware and software partitioning. Certain functionality can be either implemented in Hardware or in Software however there is usually a trade-off involved in this. With regards to the ST-080 this trade-off arose with the de-bouncing of pushbuttons and the amplification of the output waves.

We decided to go with a software implementation of the de-bouncing. This was done to reducing the costing and space effects of implementing a de-bouncing circuit. Another reason was the fact that multiple connections required de-bouncing so this duplication was more efficient in software.

To implement the output amplifier, we used a LM386 to build our amplifier. The Hardware implementation option was chosen to challenge ourselves as well as to enable ourselves to apply a greater amount our electrical knowledge to this project.

UML Diagrams

UML: Sequence Diagram

The UML Sequence Diagram shows the entities involved in the system when Drum Set is in use.

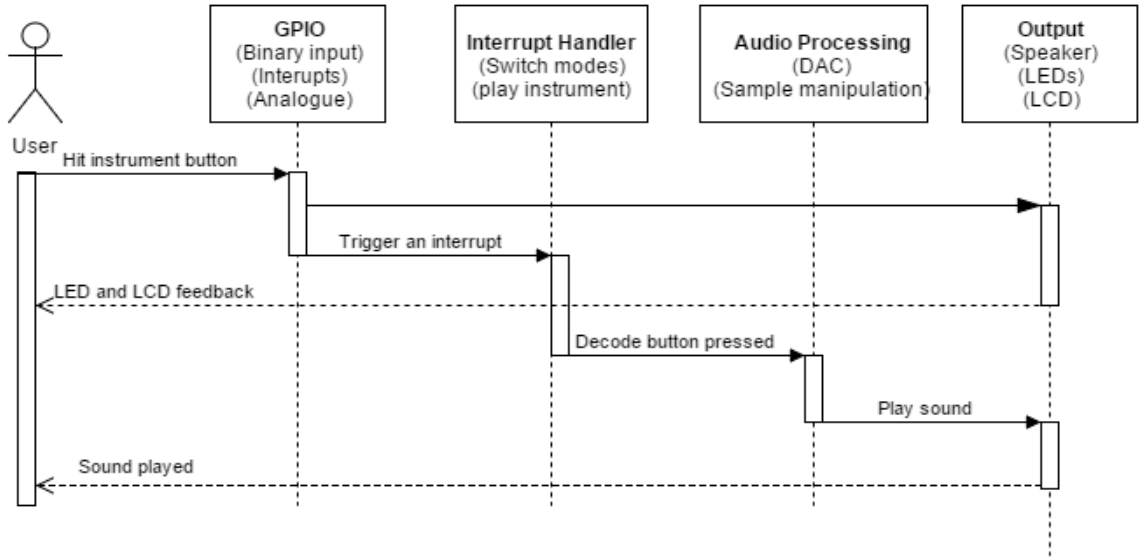

FIGURE 8 UML SEQUENCE DIAGRAM

UML: State Diagram

Figure 9 shows the behavioural State Diagram of the ST080. There are four states: Composer Mode, Freestyle, Playback Mode and Save Mode. On power on or on Reset, it the default mode is set: Composer Mode. The user can then switch between Composer, Freestyle and Playback upon selection of the corresponding mode button. While in Composer Mode or Playback Mode, the user can switch to Save Mode to store the composed song into EEPROM After the song is saved, the ST080 goes back to Composer Mode. The ST080, goes back to the Off state when power goes off.

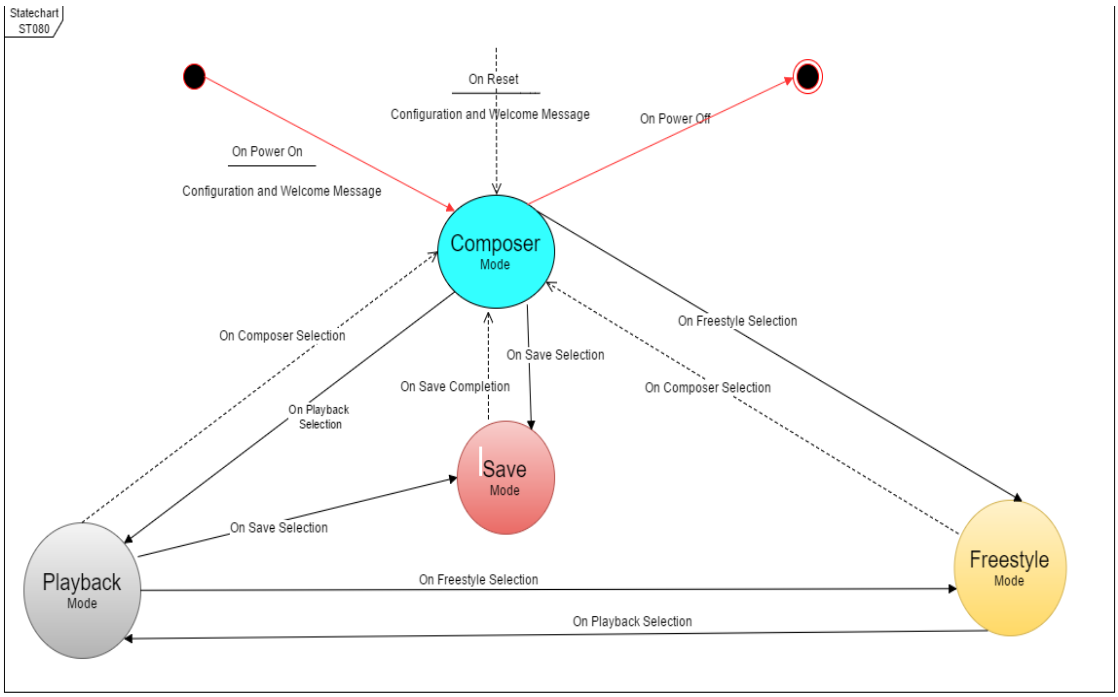

FIGURE 9 UML STATE DIAGRAM

Hardware

The ST080 case was designed to achieve the basic requirement of the design concept. In order to achieve it, it has the following specification:

- STM32F4 discovery
- One 16x2 LCD screen
- Two potentiometers (one for the tempo and one for the volume)
- Twenty-five pushbuttons and LEDs separated in four groups
    - Modes - has three pushbuttons
    - Instrument Pad - has four three Push-Buttons
    - Util - has four two pushbuttons
    - Channel rack - has four three pushbuttons
- One 64K EEPROM
- One audio amplifier
- Two speakers

- One audio connector for earphone or external speakers

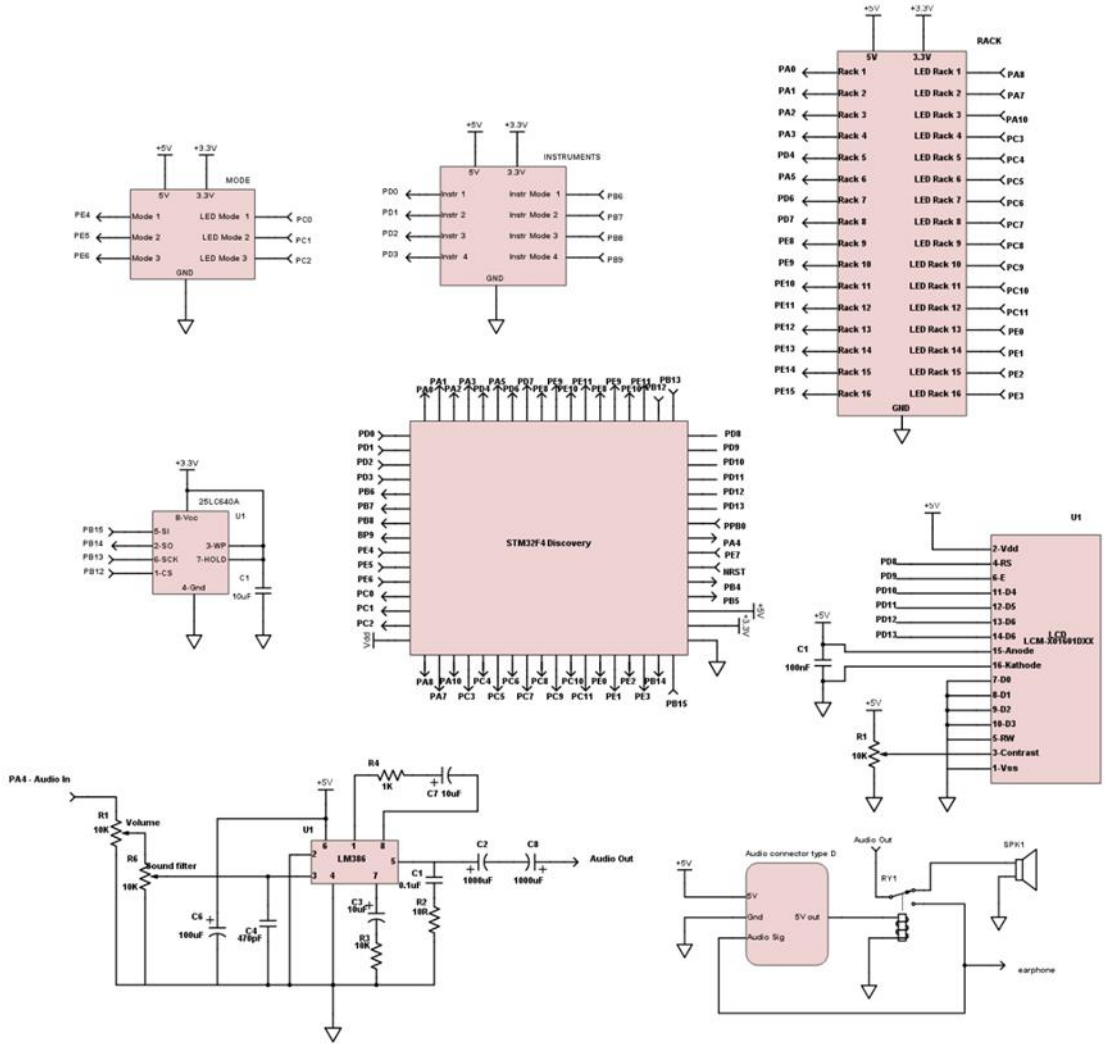

FIGURE 10 SCHEMATIC OF DRUM MACHINE

STM32F4 Discovery Board

This microcontroller is the brain of this project. This high speed, inexpensive, is perfect for the application of this project. The high speed clock that operates at up to 180MHz handles all the input

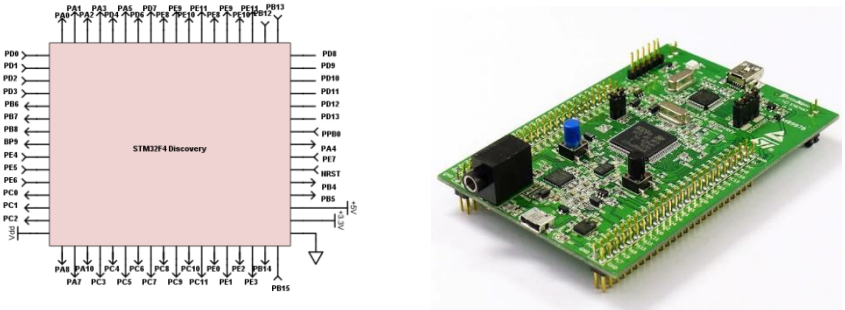

FIGURE 11 A) STM32F4-DISCOVERY PROCESSOR PIN-OUT AND B) STM32F4-DISCOVERY DEV BOARD

analysis and processing of the ST-080. A subset of the many peripherals on this board are used to fulfil the project specifications and functionality. These include the ADC, DAC, and the Timer peripherals as well as the GPIO pins. This processor is very user friendly and there are many supporting docs available to aid one.

Amplifier

The audio amplifier uses the **LM386 Low Voltage Audio Power Amplifier**. **Figure 12** below illustrates its connection with the STM.

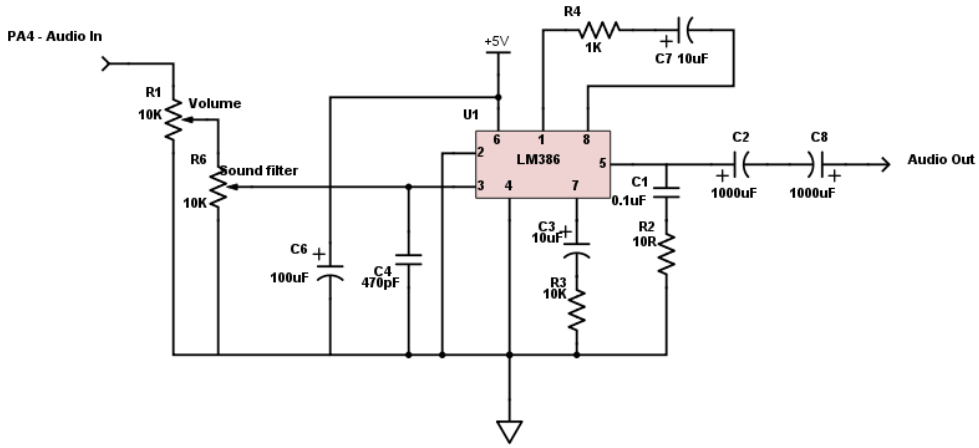

FIGURE 12 LM386 LOW VOLTAGE AUDIO POWER AMPLIFIER SCHEMATIC

It has two potentiometers, a volume and a sound filter. The sound filter is used to choose the right filtering as the audio amplifier has different performance over different audio signal fed to it. The Sound filter being set; the volume adjusts the voltage of the signal fed to the filter.

LCD Screen Interface

The LCD screen is a 16x2 display. It is set TO four-bit mode. It interacts with the micro using six pins (RS, E, D4, D5, D6, D7) and get a power of +5V.

Figure 13 illustrates its connection with the STM.

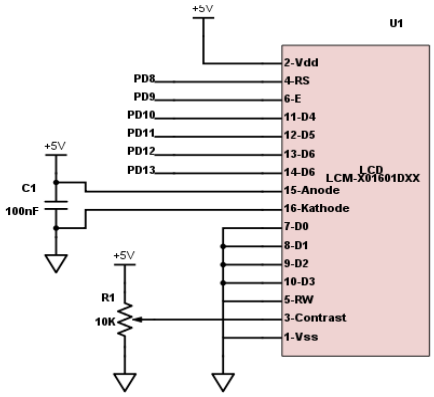

FIGURE 13 LCD SCHEMATIC

Potentiometer

Two Potentiometers were used. One was used for Tempo control and the other was used for volume control. The diagram below is for the potentiometer used for the Tempo however the potentiometer used for the volume used the same circuit. The potentiometer for the tempo is powered by 3V from the STM. **Figure 14** illustrates its connection with the STM.

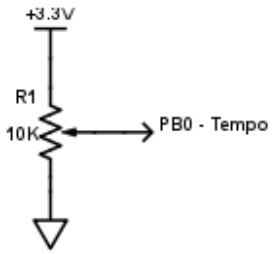

FIGURE 14 POTENTIOMETER SCHEMATIC

Internal Speakers And Audio Jack/ External Speakers

The ST080 has let the user choose what speaker the sound should be played to. **Figure 15** illustrates how it does it.

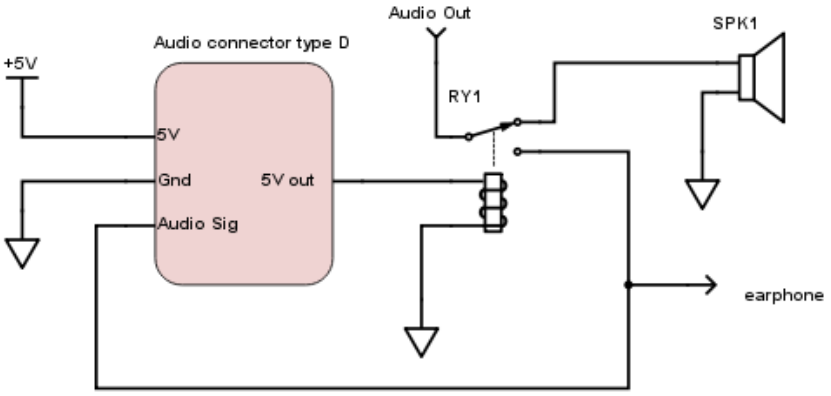

FIGURE 15 AUDIO JACK SCHEMATIC

When no audio jack is connected to the ST080, the relay is connected to the internal speakers. As an audio jack is connected, the audio connector type D send its 5V supply to its 5V out pin which switch the relay to its NO mode feeding the audio signal to the audio jack.

Channel Rack

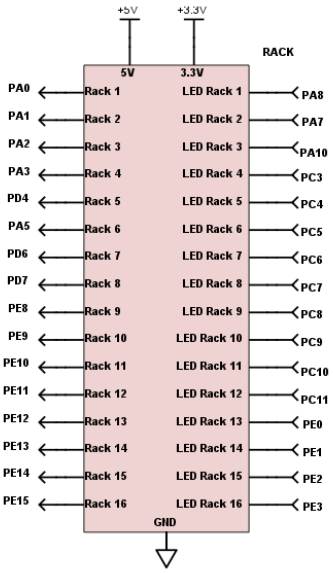

FIGURE 16 CHANNEL RACK SCHEMATIC

Eeprom Connection

The EEPROM used has 64K of memory. The diagram below illustrates its connection with the STM.

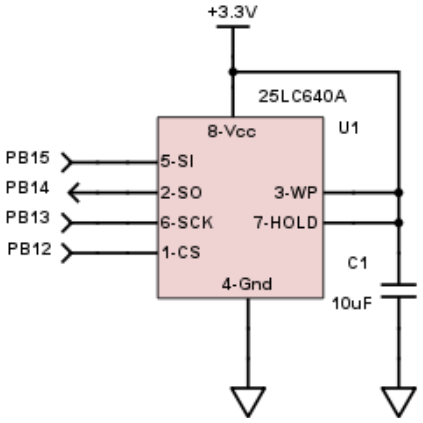

FIGURE 17 EEPROM CONNECTION SCHEMATIC

Push Buttons and LED Circuits

The pushbutton is powered by 3V line from the STM and feed that voltage it when the switch is closed (figure x). The led could not be sufficiently powered by the STM so a circuit was built to power it on a signal received from the STM (figure x).

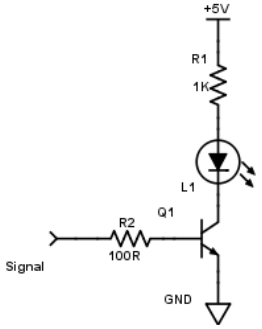

FIGURE 19 LED CIRCUIT

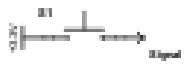

FIGURE 18 PUSH-BUTTON SCHEMATIC

Software

The diagram below (**Figure 2**) is the program's dependency graph.

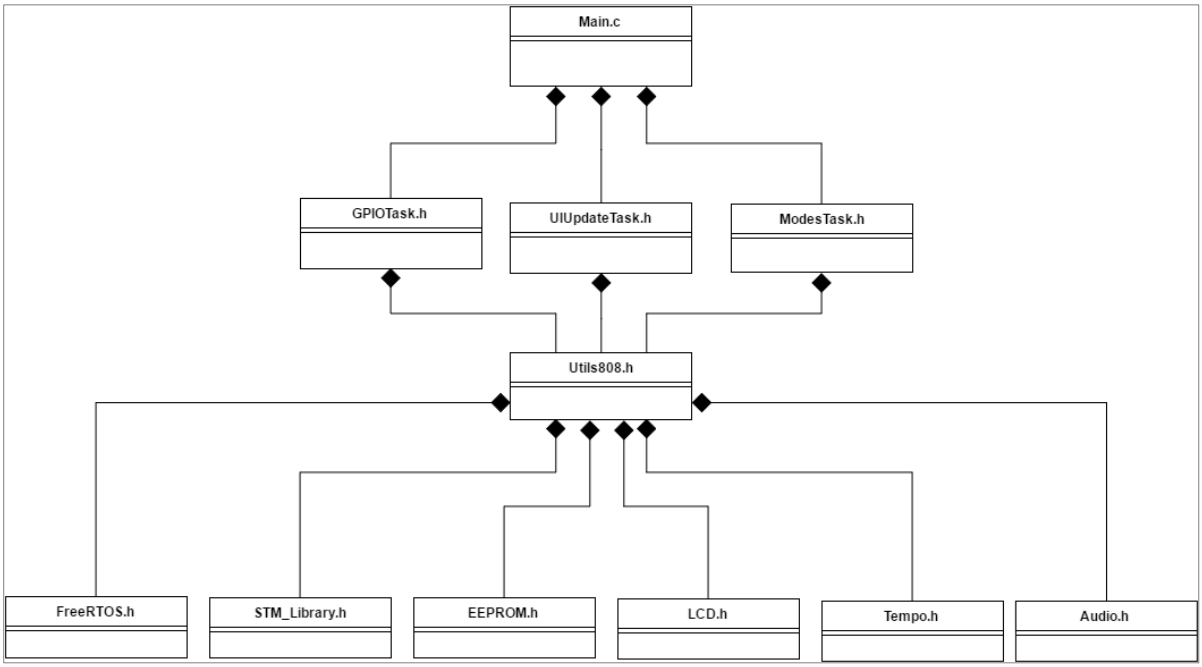

FIGURE 20: PROGRAM DEPENDENCY GRAPH CREATED USING THE HEADER FILES USED IN THE PROGRAM.

Each file in the diagram above is described in detail below.

Main.c

This is the entry point of the program. We designed it so that we have as little code in this file as possible. When the program starts, the start-up configurations function inside the Utils808.h is run. This function configures the GPIO pins, interrupts, LCD, Eeprom and ADC. Once these configs are done, the three tasks namely GPIOTask, UIUpdateTask and ModesTask, are created and the task scheduler is then run.

Utils080.h

This file is the backbone of the ST080. During the design phase of the program, the work was split into different sections and we noticed that the different functions in the STM used similar libraries. A decision was made to include all the relevant libraries, macros and global variables inside the Utils808.h file and then every other file that was created would only need to include this Utils080.h file to have access to all the libraries it needed. This also decreased code bloat inside the header files that used Utils080.

The final justification for using this sort dependency tree was that, because the three tasks in the program all shared a number of similar variables, macros and flags, the Utils080.h was the perfect place to place all these variables in.

Utils080.h therefore has the following functionality in the program:

- The libraries used in the rest of the program:
  - LCD library and GPIO interrupt libraries by Tilen Majerle [2]
  - Tempo library
  - FreeRTOS library [3]

    - Audio library
    - Eeprom library
- Global variables and flags used by the different FreeRTOS tasks.
- The channel rack array. This is a 3D array (16 x 4 x 16) with the dimensions as follows:
    - 1st dimension being the 16 different slots capable of storing 16 different songs.
    - 2nd dimension is the 4 different instruments that can be used to create a song.
    - 3rd dimension is the 16 flags representing the 16 scores on the channel rack that will be used to compose the song.
- Four arrays of the 4 different samples used on the device. These are typical samples from the **808 Drum kit** [4]; "**clap**", "**kick**", "**cowbell**" and "**open hat**" all at a sample rate of 11025 Hz.
- Arbitrary functions and types used throughout the program.

STM_LIBRARY.H

This actually refers the libraries used by the STM32 Discovery Development Board. Namely the Device Peripheral Access Layer Header File (**stm32f4xx.h**) and the definitions for STM32F4-Discovery Kit's LEDs and push-button hardware resources (**stm32f4_discovery.h**)

FreeRTOS.h

This is the library used to provide the functionality of FreeRTOS, the operating system used on the **ST080** embedded system. We decided to use FreeRTOS because we wanted the tasks in the systems were time critical (we can't afford to have lag when playing musical instruments) and making use of a Real Time Operating System meant that we would not have to worry much about task scheduling provided we use the operating system correctly.

There are three tasks in the system which take care of different things. One to manage the user interface (LEDs and LC), the second one to manage GPIO inputs and the third one to manage the logic for each mode in the system (composer, freestyle, etc…) These were defined in 3 separate files and are explained in greater detail below in the following headings.

The GPIO task has the highest priority with fairly short delays because it is critical that any input is processed with minimum lag as soon as the user presses a button.

The User Interface task has the second highest priority. We decided that this task should have a higher priority than the Mode task because we wanted the feedback to the user to be real time. The UI task does not involve a lot of computation (it is mostly just reading flags and setting GPIO pins) and also has a lot of delays in it to allow for the Mode Task to do the various tedious computation it needs to do, like adding samples. The processing done in the Modes Task does not need to be done in real time and we can afford to have some lag in this task. The only place where lag cannot be tolerated is in the Freestyle mode but this mode itself does not involve any tedious computation like adding samples.

Because the three tasks in the system use a lot of shared flags and data, binary semaphores were used to control access to these variables. All these variables were stored inside the Utils080.h file to make them accessible to all tasks. However, we had to be very careful while using them. For example, variables used in the interrupt handlers were not guarded by semaphores because if an interrupt tried to modify a variable that was locked, the system would be deadlocked since interrupts cannot be pre-empted by FreeRTOS tasks. We also decided not to guard any variables used by the UI class. The reasoning behind this decision was that, because the UI is refreshed very frequently, stale variables would not have much of an impact on the user because the user simply would never be able to notice the change.

However, locks were used for some variables like the array for the channel rack flag and other status flags used in the composer and playback mode since access to these variables had to be controlled.

ModesTask.h

This file includes the definition of the task that will provide the functionality of the different modes in the system. The modes are as follows:

- **Composer Mode:** This is the mode that allows the user to compose the song. The flags stored inside the channel rack array (inside the Utils080.h) will be used to add the different instrument samples to create one large array (we will call this **DAC_Buffer** and it is stored inside Utils080.h) which will be looped as the users continues to modify the song. When a flag on the channel rack is high, the sample would be added to the DAC_Buffer at the corresponding position.
- **Playback Mode**: This mode allows the user to listen to any of the 16 different songs stored on the ST080. It works in the same way as the composer mode where it adds samples to the DAC_Buffer using the flags in the channel rack array but the user would not be allowed to modify the flags. Instead, the user can use the 16 pins on the channel rack to listen to the 6 different songs stored on the ST080.
- **Freestyle Mode**: This mode allows the user to play any of the instruments with no restrictions on tempo or structure (hence freestyle
- **Save Mode**: This mode will save the modified songs onto the device's permanent storage. Any songs that are not saved will be lost when the ST080 resets. When the ST080 starts up, it loads the channel rack stored in the EEPROM into RAM. When the save button is pressed, the ST080 will copy the channel rack from RAM to EEPROM. We decided to do this instead of modifying the channel rack straight from EEPROM because reading from EEPROM is much slower than using RAM and this also allows the user to discard any changes they make to their music by simply resetting the device.
- **Error Mode**: This mode will be switched to when the program encounters any exception which cannot be recovered from (e.g. malfunctioning pins, peripherals, etc.) This mode will essentially brick the device until it is reset.

EEPROM.h

This library allows for storing and reading data from the eeprom.

Audio.h

This library manages audio playback in the ST080. For Composer and Playback modes, it loops through **DAC_Buffer** array (stored inside Utils080.h) and uses the DAC to output the samples onto a GPIO pin. A custom external amplifier is used to produce the sound.

For the freestyle mode, instead of looping through an array, when a sample is played, the samples for that instruments are passed through the DAC and pushed to the external amplifier.

How tempo was implemented

The ST080 can play music at different tempos and it does so without distorting the actual sound of the instruments. Instead of playing the samples at different rate (which would result in varying pitch), the **DAC_Buffer** is split into 16 smaller arrays and a timer is used to play these 16 different arrays. The rate at which the timer plays these arrays is set by the tempo, which is obtained using the **Tempo.h** library.

GPIOTask.h

This task is used to read the GPIO pins on the ST080. In the save, composer and freestyle mode, the 16 channel rack pins are read and flags inside the Utils080.h (including the channel rack array) to allow the other tasks to act on the detected changes. No GPIO pins need to be read in Freestyle and Error mode.

UIUpdateTask.h

This task manages the User Interface for the program. It updates the LCD and LEDs depending on the mode. There are 4 LEDs for the main modes: Composer, Playback, Freestyle and Save. The current mode is detected using the flag MODE set in Utils080.h. Then, the correct LED is turned on using a static GPIO and Pin mapping.

While in Composer Mode, the LEDs of the channel rack are updated for the current instrument depending on the buttons pressed by the user. The decision to turn on or off is based on shared flags for the channel rack that is updated in GPIOTASK.H. The LCD is similarly updated to display the current instrument being updated, the tempo, and the current mode.

In Playback Mode, the channel rack LEDs are continuously cycled through as the song is being played. It is implemented using the tempo at which the song is being played at. Moreover, since the channel rack buttons are used to select the saved song that user wants to play, the corresponding LED is turns on to shows the selected song. Besides the LEDs, the LCD not only displays mode but it also tells the song being played and in case the user selects an empty song, it also notifies that no song had been saved there.

In Freestyle mode, the LCD just displays the Mode and tells the user to enjoy its freestyle section. There is also one LED for each of the 4 instruments. The correct LED is turned on for a short period as the user plays to indicate the instrument being played. By so doing, it creates an interesting light effect. Each of the 4 instruments has a flag that is set in the interrupt handler when the corresponding button is pressed. These flags are used to detect which LED needs to be turned on or off.

In Save Mode, the save LED and the LDC are both used to indicate that the ST080 is in Save Mode and give instructions (the LCD only) to the user as to where the song is to be saved. In general, the flags are continuously checked to detect any change and subsequently update the LEDs and/or the LCD.

The sections of the 16x2 LCD are updated dynamically only when we change mode, instrument, song or tempo to avoid unnecessary computing and most importantly blinking.

LCD.h

This library is used to write to the LCD screen.

De-bouncing

Software de-bouncing was implemented to ensure that only valid button presses caused instrument changes or interrupts. This was done by having a previous button press time as well as a current button press time. If these times are too close to each other than that specific current button press would be ignored.

Experiment

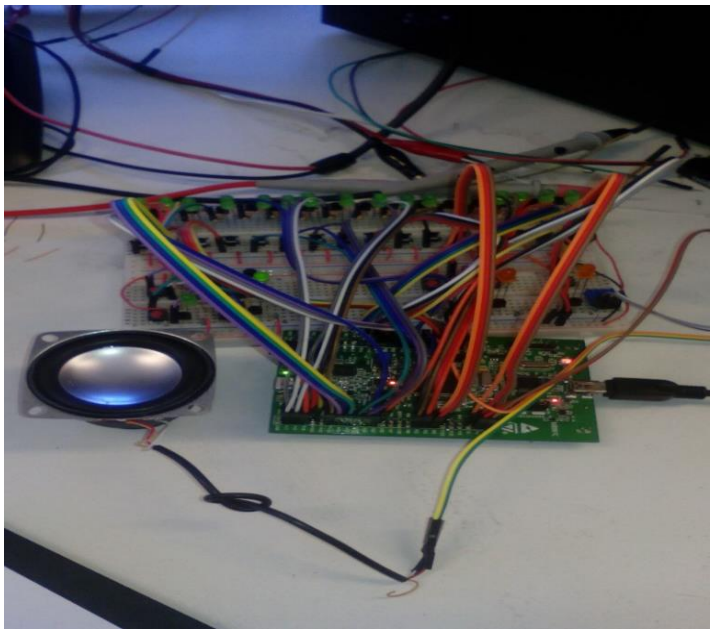

FIGURE 21 EARLY TESTING STAGES

Software Testing

Because most of the software was developed without access to the hardware, there were many challenges to test the functionality of the software. The different methods listed below were used in both white-box and black-box testing (without the hardware):

- Some of the results were copied from the running program and analysed using other software to confirm the functionality. For example, to verify that the samples were being added properly, the debugger was used to stop the program after adding the samples. These samples were copied into MATLAB and a graph was plotted. This graph was compared to other graphs that were plotted when the samples were added using MATLAB. To check for distortions, Audacity was used to plot the graphs of samples after being manipulated in the program.
- Because we did not have an amplifier and speaker to listen to the sound produced by the DAC, we used oscilloscopes to plot the signal coming out of the pins and compare it to the graph produced by MATLAB for the same signal.

FIGURE 22 OUTPUT OF AUDIO AMPLIFIER

- The on-board LEDs were used to check if the different modes were running properly. They were set to blink at different rates and periods depending on what was being checked. The LEDs were also used to check if pins were responding fast enough when pressed.
- The oscilloscope was also used to check if there was too much lag in the system due to the adding of different samples into the large arrays.
- Wires were used to implement button pressing.
- The debugger was set to stop at different breakpoints and variables were manually checked to see if they were what they were supposed to be.

Hardware Testing

Circuits

The circuits were built as modules. Each of them was tested separately. **Figure 22** above is the output of the audio amplifier.

Final case

After each module was tested, they were connected together in the case. A program was written to test each module interacting with the STM. The code can be found on github --link--. The program writes something to the LCD, load a sound depending of a value loaded from eeprom (the value is changed every time so different sound are played on power), toggle the led corresponding to the pushbutton if it is pressed. When mode one is pressed, one sound is played. When mode two is pressed, the sound is played depending on what tempo is set from the pot.

Mechanical Case

The mechanical case was entirely designed in SOLIDWORK, then laser cut on a wooden plate. **Figure 23** depicts the different views of the case.

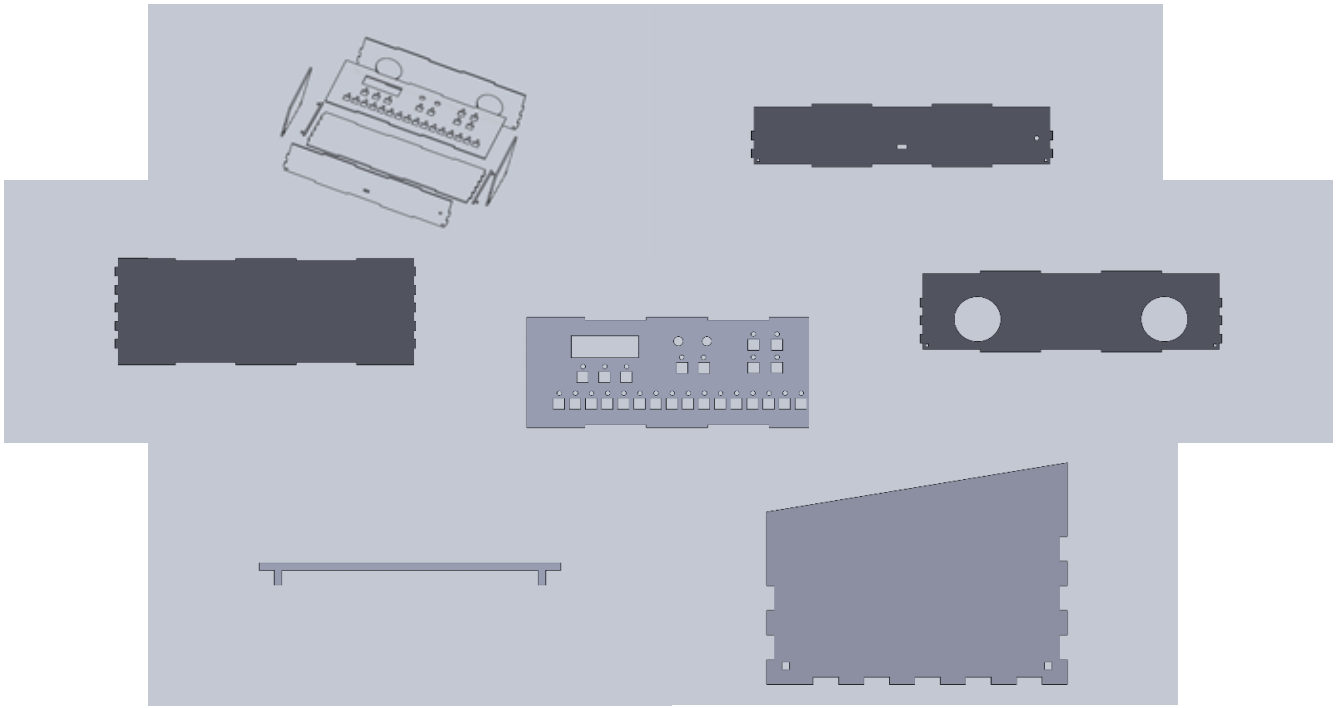

FIGURE 23 MULTIPLE DESIGN VIEWS OF THE DRUM MACHINE HOUSING COMPONENTS

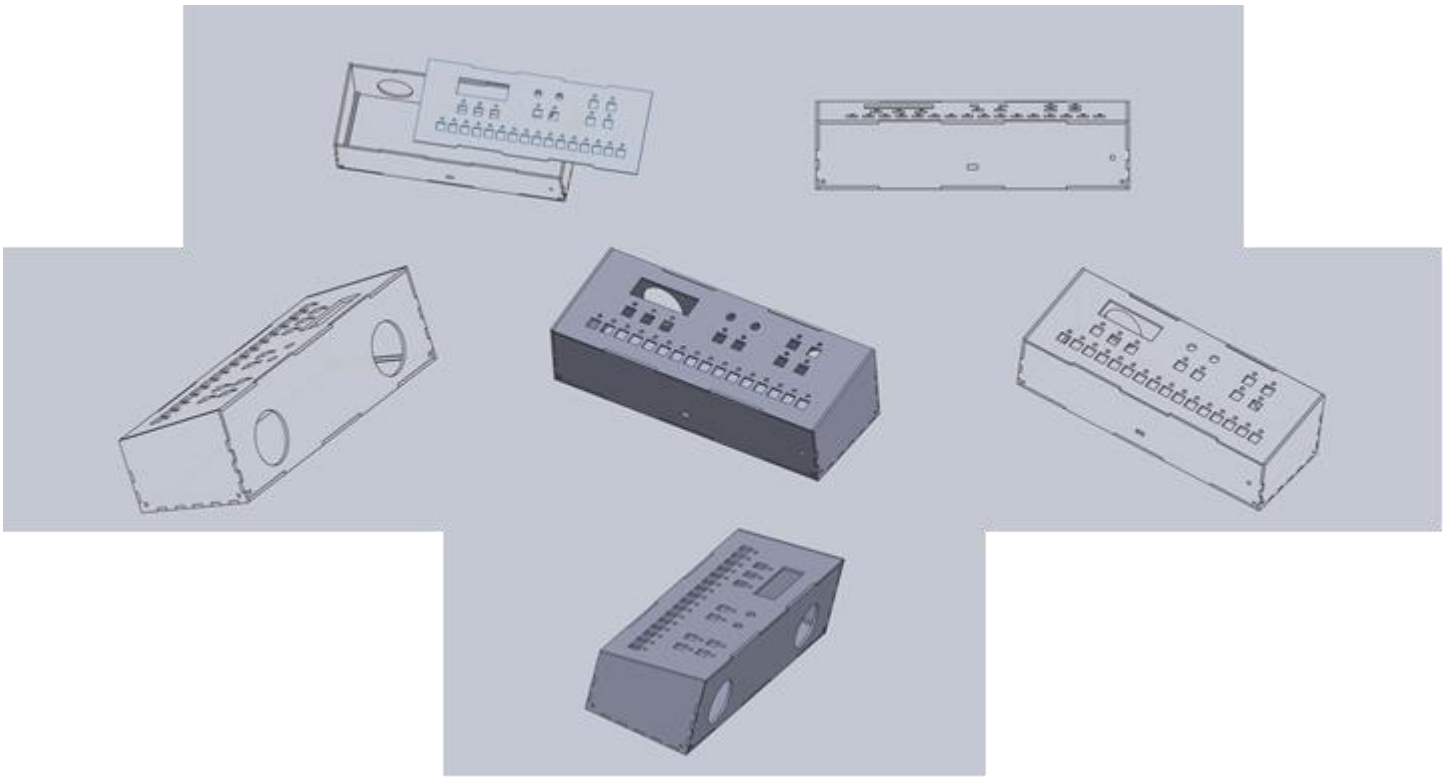

FIGURE 24 MULTIPLE DESIGN VIEWS OF ASSEMBLED DRUM MACHINE HOUSING

Integrated Testing

The following testing methods were used during the hardware and software integration:

- Each button was pressed to check if it responded as expected on the STM (black-box)
- Breakpoints were set in the code to check if the right branches were being taken when the different buttons were pressed (white-box).
- Different people were asked to use the ST080 and give feedback on their experience. Issues they helped us improve were the mostly on the user interface (messages to display, LEDs to light up, duration of LED blinking and sounds to use as the instruments)
- The sound output was listened to for any anomalies and changes were made in code or in the amplifier to improve the sound.

Results and Conclusions

This document has achieved the purpose of laying out the basic structure of the Drum Machine. The Software and Hardware requirements have been recorded and the initial design has been decided upon. Moving forward, the prototype will be built and structured according to the specifications laid out within this document.

Challenges Faced

Many challenges were faced during this project and these are listed below:

- Features dropped when implementing the prototype:
    - **Tutorial**: We decided that the tutorial feature was not necessary to implement on the ST080. Instead, a tutorial video will be created along with a manual. This frees up more space to use on the ST080.
    - **Analogue filters**: There wasn't enough time and human resources to design, implement and test the analogue filters. We put this feature on hold to implement if we found enough time at the end of the project but this was not possible.
- **Integration**: Because we had partitioned the hardware and software, integration was challenging since it was not possible to test the software without the actual hardware and vice versa during development. A lot of bugs also creeped up during integration and because the software team did not have in-depth knowledge of how the hardware worked (and vice versa), some simple bugs would take a while to fix (for example, forgetting to reset or check certain flags set by the hardware)

Possible Improvements

There are many ways this project could have been improved. These are listed below:

- Implement Different drum-kits allowing for multiple sound types and music styles.
- Implement a tutorial interface that would basically teach users how to play or compose different songs or beats (e.g. "Twinkle-Twinkle little star")
- Add Analogue filters that would allow for more variety of sounds with the same instruments
- Implement external memory to all for more music storage capabilities. Could have background music in addition to the beats.
- Add a recording feature to record music and store it.

References

[1] [Online]. Available: https://www.google.co.za/search?q=Drum+machine+plan&source=lnms&tbm=isch&sa=X&ved=0ahUKEwjA456XzKrOAhVpJsAKHeeDCzsQ_AUICCgB&biw=1920&bih=955#tbm=isch&tbs=rimg%3ACdcuHxhDgO7iIjjLHBPapUQPWs69jiPTQv6g1qNRR--26CNjsFgHEISp9ECJX0YpkOYFCXbGuwBl1AARUFCw47I12SoSCc.

[2] T. Majerle, “stm32f4-discovery.com,” [Online]. Available: http://stm32f4-discovery.com/2014/06/library-16-interfacing-hd44780-lcd-controller-with-stm32f4.

[3] R. T. E. Ltd, “FreeRTOS,” [Online]. Available: http://www.freertos.org/.

[4] hipstrumentals.com, “808 Mafia Drum Kit (Drumkit),” [Online]. Available: http://www.hipstrumentals.com/2014/09/808-mafia-drum-kit-drumkit/.

Appendix

TABLE 2: FULL BILL OF MATERIALS

RefDes	Name	Value	Manufacturer Part Number	Quantity
S1	Normally Open			25
R1	RESISTOR	1K		25
R2	RESISTOR	100R		25
Q1	NPN			25
L1	LED			25
U1	Audio amplifier	Value	LM386	1
R2	RESISTOR	10R		1
C1	NON POLARIZED	0.1uF		1
C2	POLARIZED	1000uF		1
R3	RESISTOR	10K		1
C3	POLARIZED	10uF		1
C7	POLARIZED	10uF		1
C8	POLARIZED	1000uF		1
R4	RESISTOR	1K		1
C6	POLARIZED	100uF		1
R6	VARIABLE	10K		1
R1	VARIABLE	10K		1
C4	NON POLARIZED	470pF		1
R1	VARIABLE	10K		
U1	LCD	Value	LCM-X01601DXX	1
C1	NON POLARIZED	100nF		1
R1	VARIABLE	10K		1
U1	64K EEPROM	Value	25LC640A	1
C1	NON POLARIZED	10uF		1
RY1	RELAY, SPDT			1
U1	Audio connector type D	Value		1
SPK1	SPEAKER			1