
# Software Decomposition

*Team 3A*

*Kennedy Houston*
*Andre Mbakwe*
*Aaron Hoffman*

**Date: January 29, 2021**

# Revision Record

| Date | Author | Comments |
|---|---|---|
| Jan 29, 2021 | Team | Document Created |
| Jan 29, 2021 | Team | Software Documentation added |
| Jan 31, 2021 | Team | Software Documentation updated |
| Jan 31, 2021 | Andre M | Software State Machine and Architecture updated |
| Feb 2, 2021 | Aaron H | Block Diagram updated (consolidated digital UI into the controller) |
| Feb 2, 2021 | Aaron H | Added sub-system input/output table |
| Feb 3, 2021 | Kennedy H | Updated Overview sketch and project description |
| Feb 3, 2021 | Aaron H | Updated system design description |
| Feb 3, 2021 | Andre M | Added team roles |
| Feb 5, 2021 | Andre M | Design sketch updated |
| Feb 8, 2021 | Andre M | Updated Sub-system description (controller) |
| Feb 18, 2021 | Aaron H | Updated Block Diagram and Sub-System Input/Output table |
| Mar 10, 2021 | Aaron H | Added subsystem block diagrams |
| Apr 02, 2021 | Aaron H | Added PCB and schematic board |
| Apr 05, 2021 | Andre M | Updated software design |
| Apr 07, 2021 | Andre M | Added integration design |
| Apr 12, 2021 | Kennedy H | Updated mechanical design and conclusion |

# Table of Contents

Generating Table of Contents for Word Import ...

## System Overview

The CelloBot project will feature a cello shape with a body that rotates on a base and a rotating wheel which moves a bow back and forth across its front strings. The on/off button will be on the base of the structure, and all hardware will be stored inside. The purpose of the robot is to play the tune of "Row, Row, Row Your Boat" through a speaker in harmony with a conductor tune which plays preliminary notes to set tempo and octave. The device will calibrate to the tempo set by the conductor and keep time with other orchestra members.

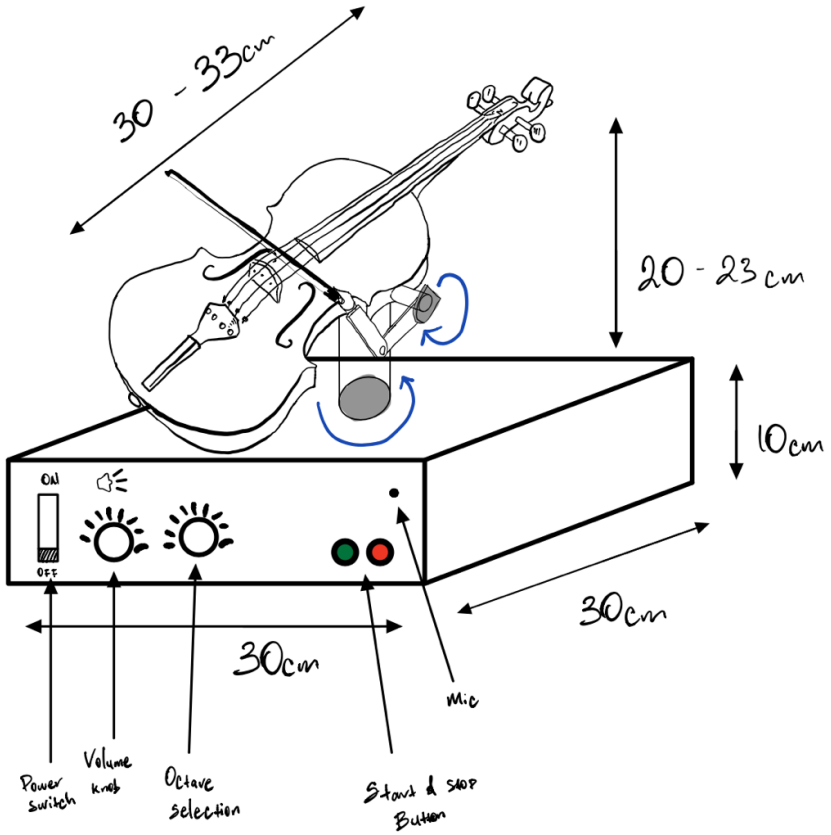


*Figure 1: Robot design sketch*

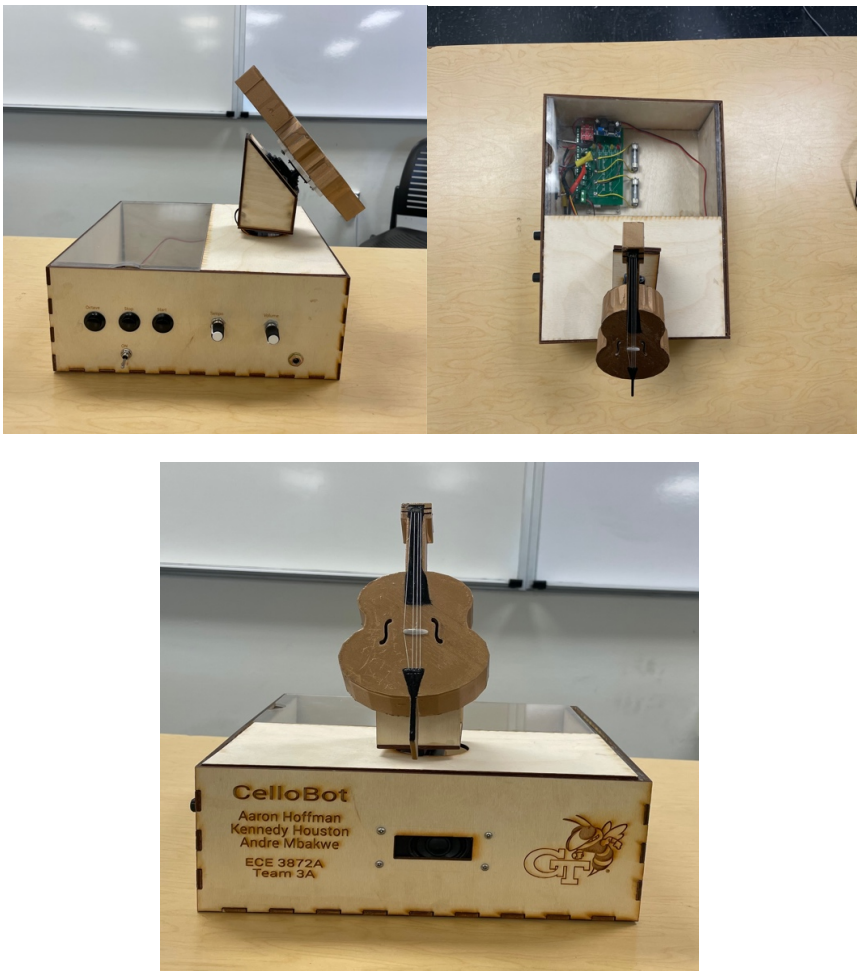


*Figure 2: Final Robot design sculpture*

## System Design

The inputs of the system include an analog audio source, 120V AC power, and digital input from the user. The system outputs are mechanical motion of the robot and audio of the children's song, "Row Row Row Your Boat". The sub-systems include power, mechanical, Audio, and controller blocks. The power block includes a 120V AC/DC converter. The mechanical block includes two servos. The audio block includes 0.5W speaker, an audio amplifier circuit, and a volume adjustment knob. The controller block contains an Arduino Uno, a digital start/stop switch, tempo adjustment knob, and a frequency adjustment switch.

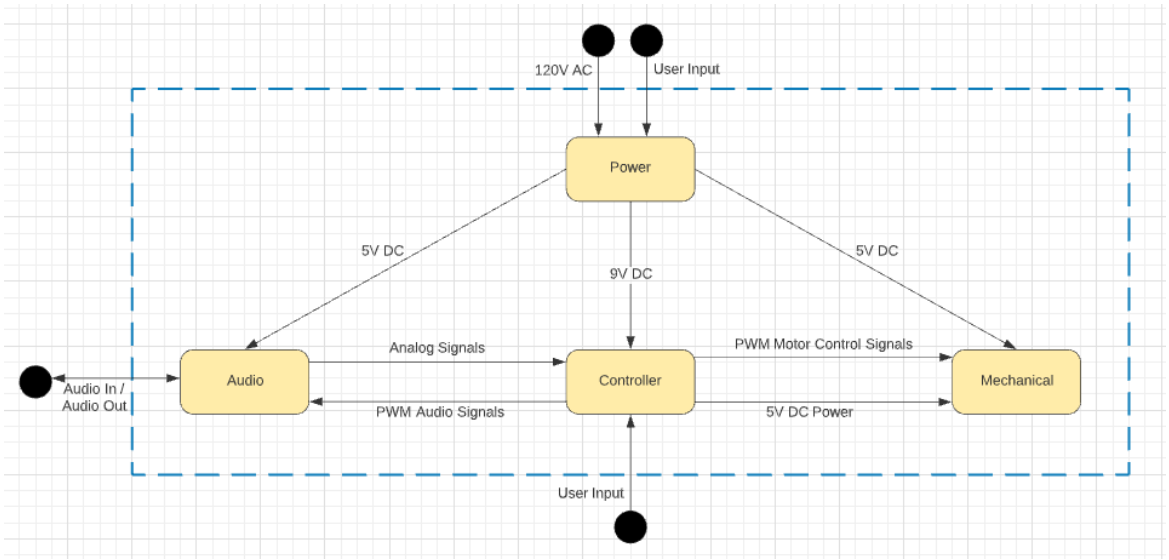


*Figure 3: System Diagram*

*Table 1: Sub-System and Inputs and Outputs*

| Interface | Source | Destination | Description |
|---|---|---|---|
| 120V | System Input | Power | 120V AC Power |
| 9V | Power | Controller | 5V DC Power |
| 5V | Power | Audio | 5V DC Power |
| | | Mechanical | 5V DC Power |
| 3.3V | Controller | Mechanical | 3.3V DC for motor/servo |
| Motor Control Signals | Controller | Mechanical | PWM signal output (servos) |
| PWM Audio | Controller | Audio | PWM signal output (song) |
| Analog Signals | Audio | Controller | Analog signal from microphone |

## Sub-System Design

Our design includes four different subsystems: controller, power, audio, and mechanic. The controller will integrate all subsystems by converting the user input to signals for the sound and movement systems. The power sub-system will convert 120V AC to 9V DC and 5V DC. The audio sub-system will process the raw tone signals and convert it to sound, and the mechanical sub-system will execute the motion of the cello figure.

## Power Sub-System

The power sub-system takes in 120V AC which is converted to 9V DC through a 20W transformer. Power is then routed through a switch controlled by the user. Next, power is routed through a 1 Amp fuse before distributing 9V DC to the microcontroller while another branch steps down to 5V DC through a buck converter. The 5V DC is then passed through a 0.75 Amp fuse before powering the audio and mechanical subsystems.

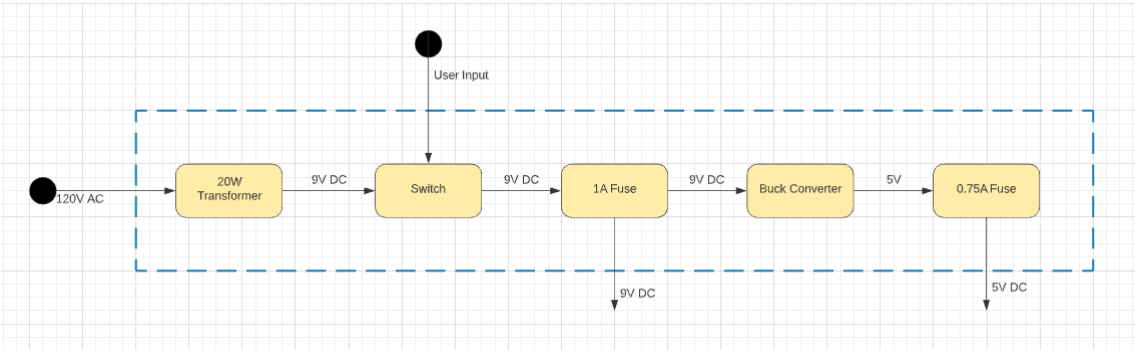


*Figure 4: Power Sub-System*

## Controller Sub-System

The controller subsystem will take in 9 Volts DC as input. The Controller will use read input signals from the pushbuttons and potentiometer. It will send out digital output signals for the servo motors and state LEDs, and 5V signals for the audio digital analog converter.

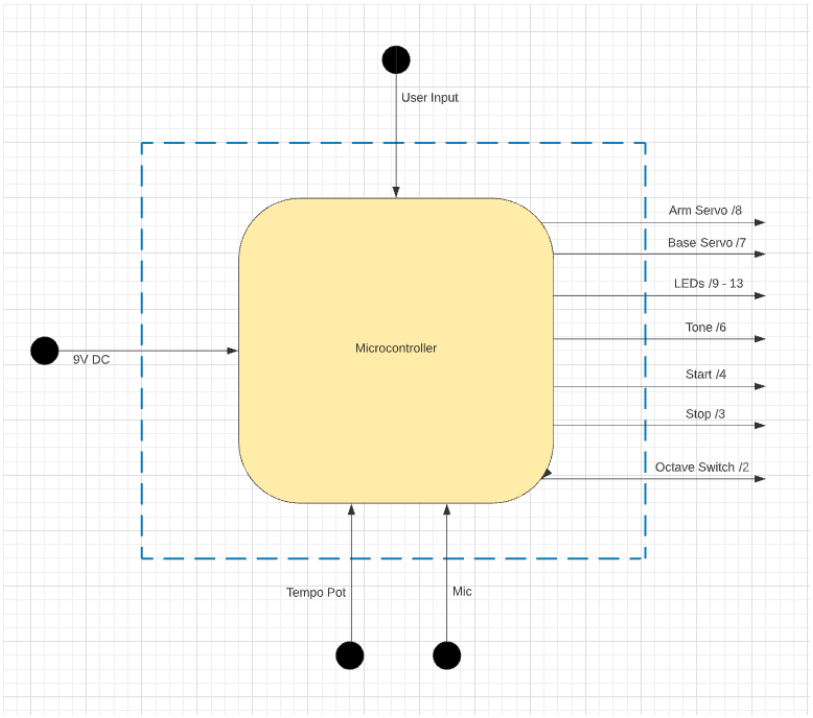


*Figure 5: Controller Sub-System*

## Audio Sub-System

The audio sub-system has a 5V DC input and digital tone inputs which are processed through an audio amplifier. From the amplifier, the signal is transmitted through a speaker to music. The volume is controlled through a potentiometer connected to the amplifier.

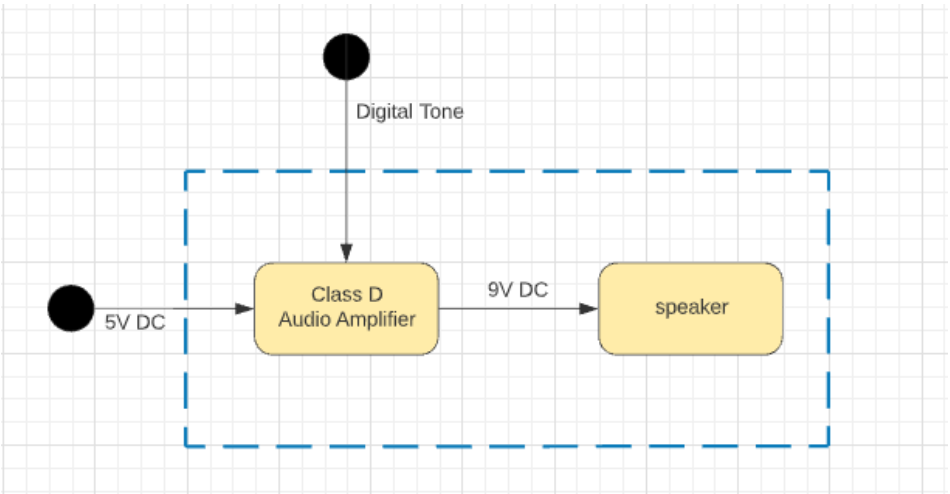


*Figure 6: Audio Sub-System*

## Mechanical Sub-System

The mechanical sub-system is composed of two servos which are powered by a 5V DC signal. The servos are controlled through digital inputs from the controller which dictates their movement patterns.

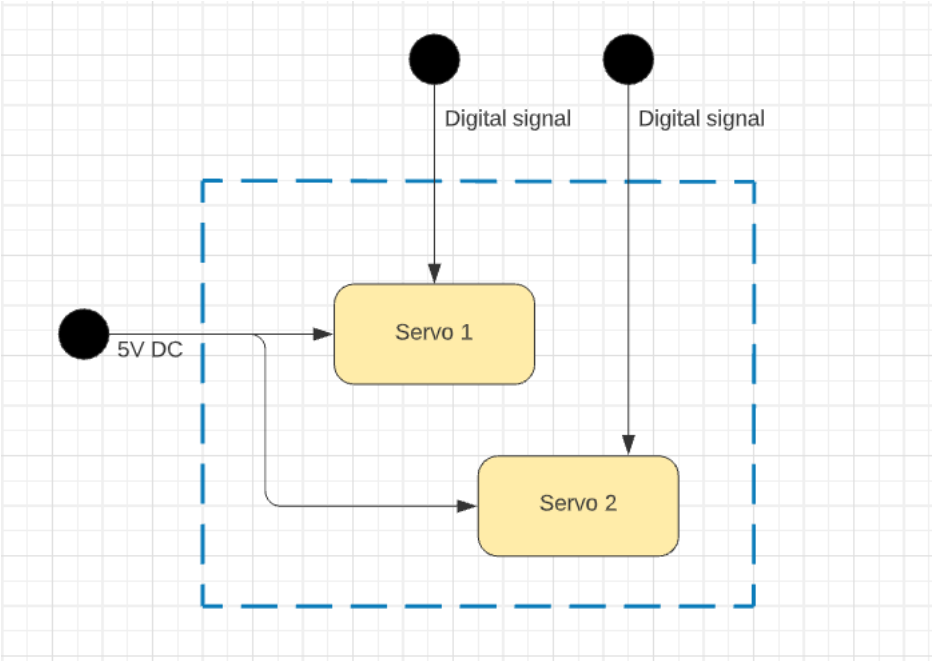


*Figure 7: Mechanical Sub-System*

## Electronic Design

The electrical design is composed of an Arduino Uno microcontroller, a class D audio amplifier, a buck converter, speaker, and two servos which were connected to a custom PCB. A block diagram of the system and user controls is shown below, along with schematics for the PCB, the second of which contains corrections that were required to get the system to function properly. The block diagram contains all the key elements of the final system, including the user interfaces used to controller the robot.

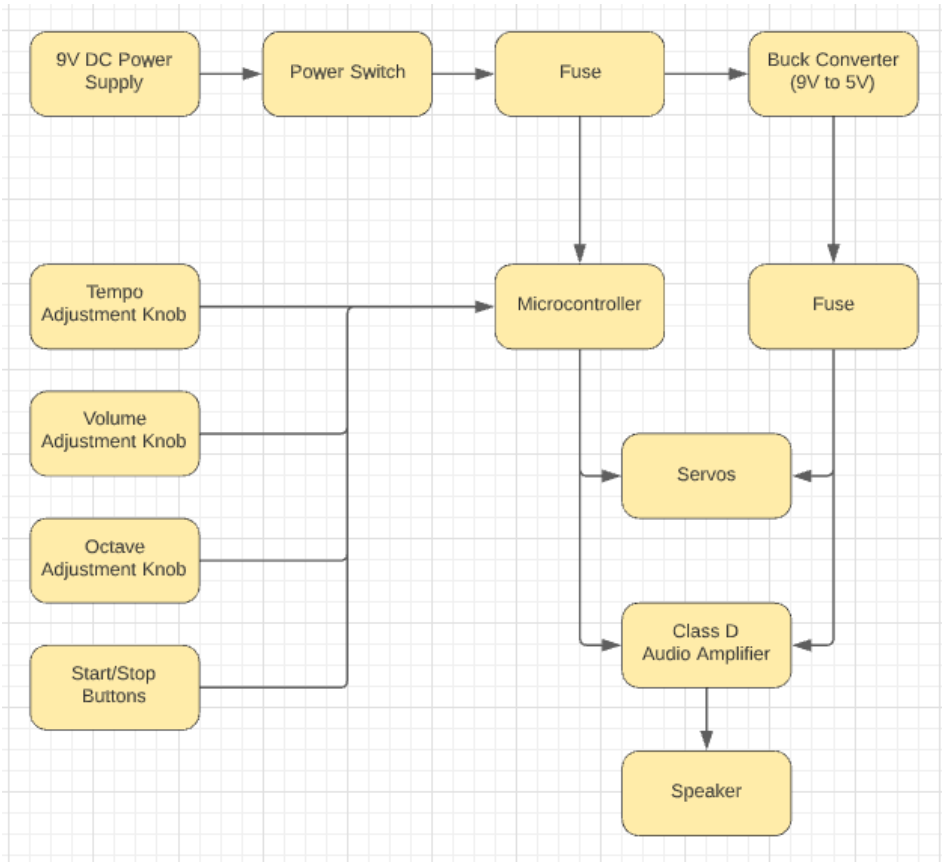


*Figure 8: Electrical Component Block Diagram*

## PCB Design

The block diagram was used to design the PCB schematic below. Eagle software was used for this process to implement the four sub-systems: controller, power, audio, and mechanical. The controller will integrate all subsystems by converting the user input to signals for the sound and movement systems. The power block will convert 120V AC to 9V DC and 5V DC. The audio block will process the raw tone signals and convert it to sound, while the mechanical block will execute the motion of the figure.

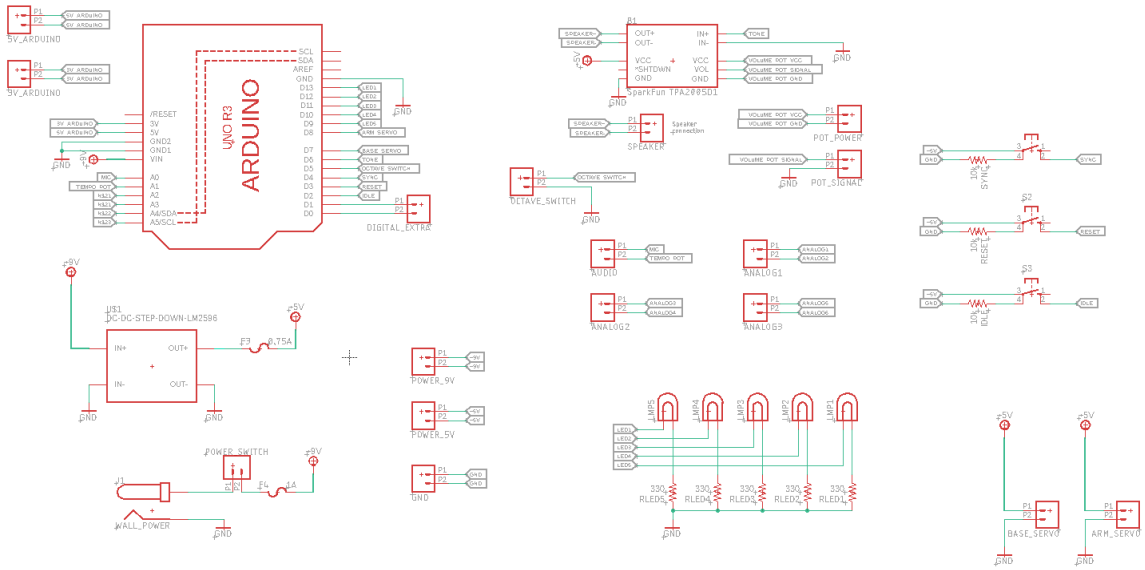


*Figure 9: PCB schematic created in Eagle. This design incorporates an Arduino Uno, buck converter, audio amplifier, LEDs, two servos and a speaker.*

This second schematic reflects corrections that needed to be made after the board was printed. First, the control pins for the servos were not run to any junctions and needed to be white wired. Second, the buttons were wired incorrectly such that the input always ran high, so these pinouts were corrected. Last, the octave switch was moved to one of the buttons and the sync and reset buttons were reprogramed to start and stop Cellobot so their names on the schematic were changed to reflect their new functionality.

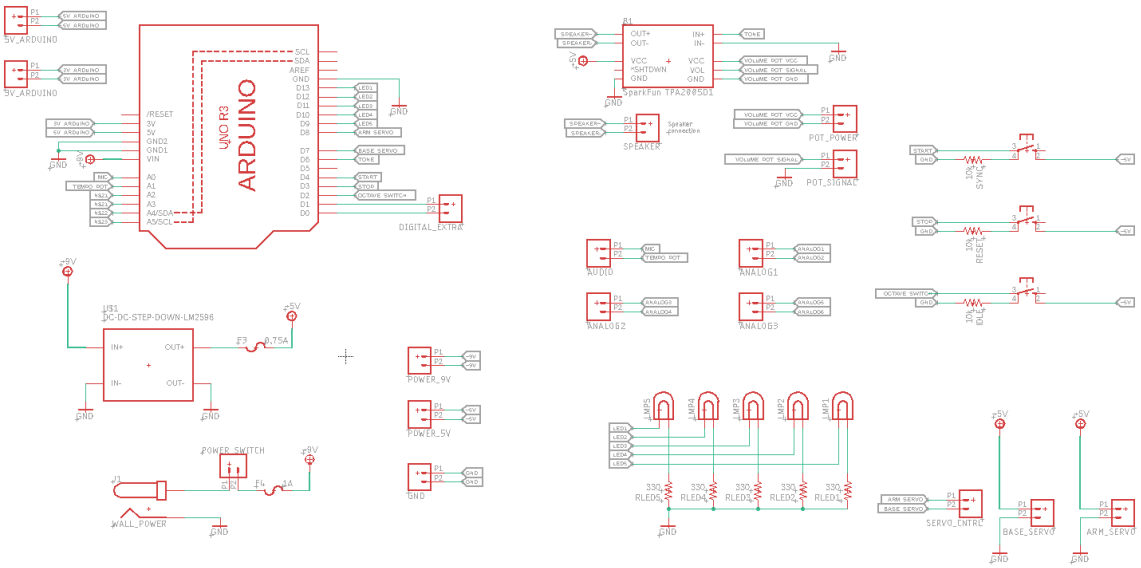


*Figure 10: Updated PCB schematic created in Eagle reflecting corrections that were required after the board was printed by white wiring new connections.*

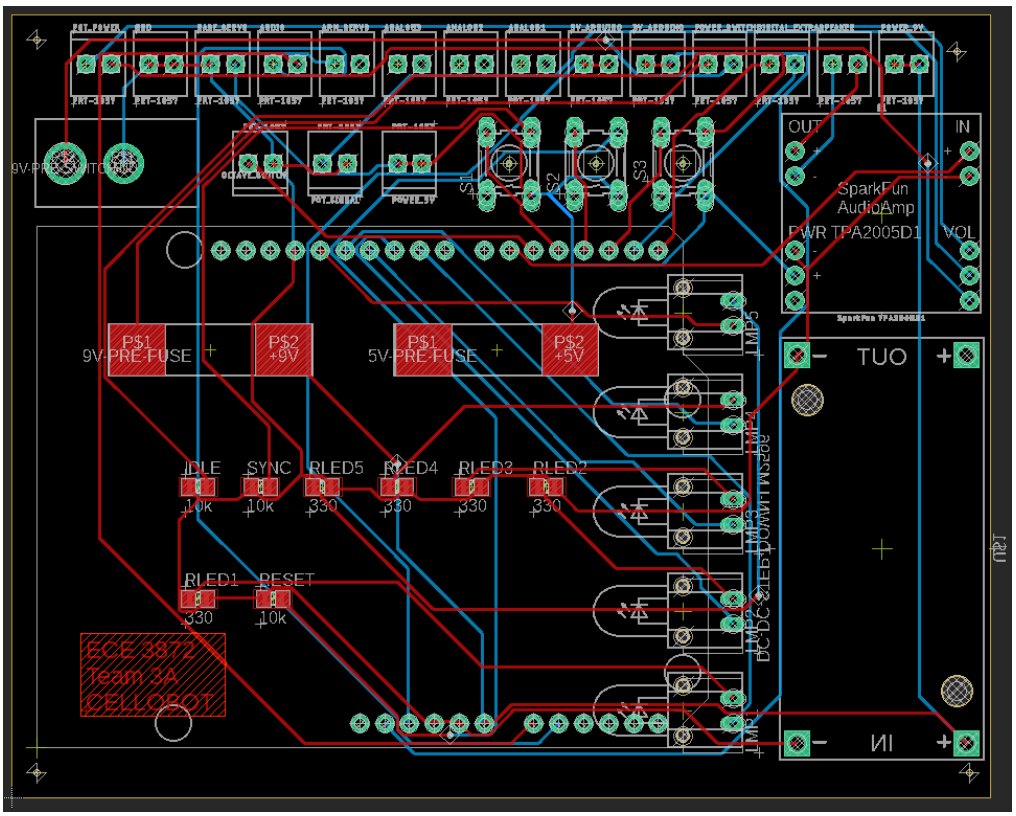

*Figure 11: PCB board view of the Eagle schematic. This board connects directly to the pins of an Arduino Uno and includes footprints for a buck converter and audio amplifier.*

## Mechanical Design

The mechanical design of the CelloBot centers around two servos which are attached to a model of a cello and control two dimensions of movement. The cello is a 3D printed model, shown in figure 12.

*Figure 12: OBJ file of the cello to be 3D printed and mounted.*

The foundation of the project features a wooden box that contains the printed circuit board, connections to the user interface dials and buttons, and the servo that swivels the cello on its base. Additional holes were drilled into the side of this box for the user interface dials and buttons.

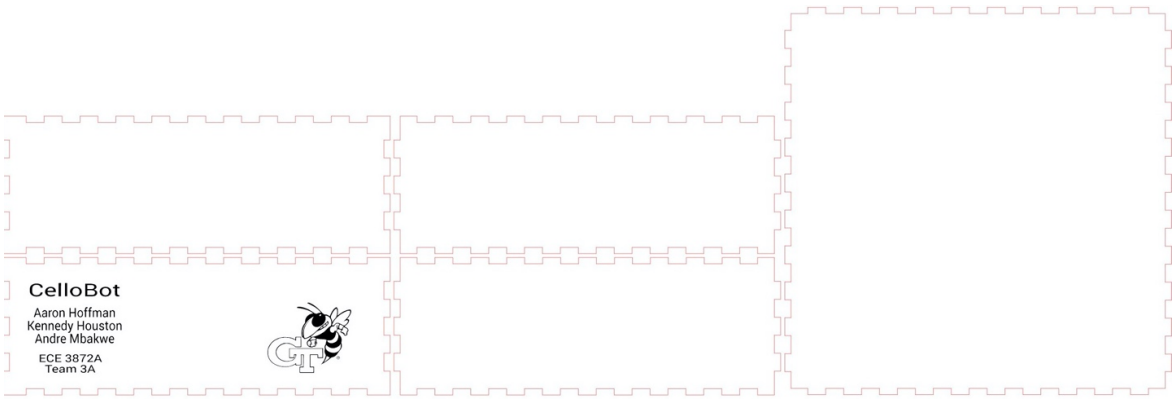


*Figure 13: SVG file used to laser cut the foundation box for the CelloBot.*

*Figure 14: Laser cut wooden box with uncut wooden panel and acrylic pane for lid.*

There is a wooden casing that holds the cello and the smaller servo that controls the rotational movement of the cello. This wooden casing is mounted on top of the wooden box and is attached to the larger servo, thus allowing the entire casing to rotate via this larger servo.

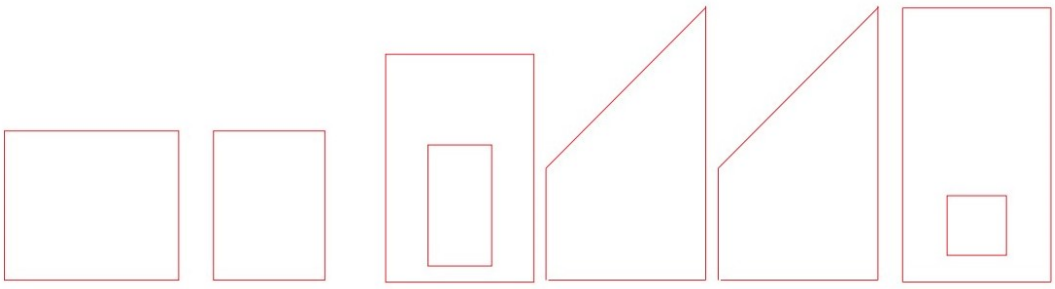

*Figure 15: SVG file used to laser cut the wooden casing that holds the smaller servo and the cello.*

The box also features a pane of acrylic and a pane of wood that close the box on the top. The wood has a hole cut into the middle to make room for the larger servo. As aforementioned, the wooden casing sits atop this larger servo. Figure 13 shows a CAD model of the entire structure.

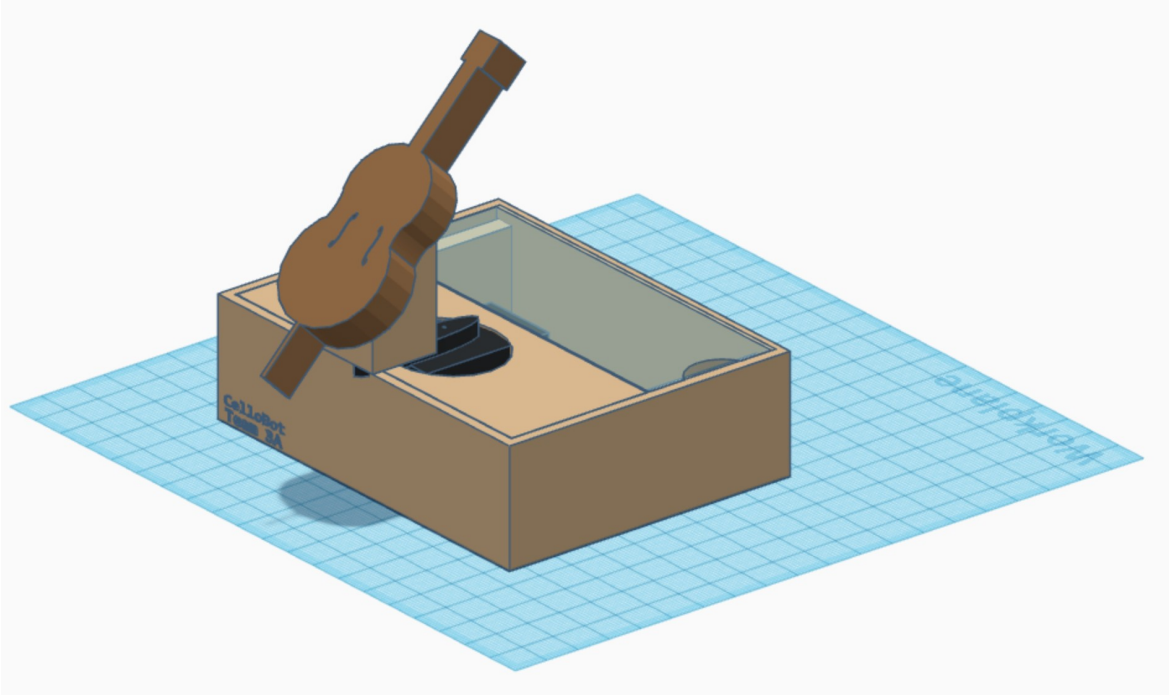

*Figure 16: 3D rendition of the mechanical design and mounting. There is a removable pane of acrylic that allows for viewing and manipulating the contents of the box if necessary.*

## Software Design

We have two software designs, one includes 6 different states – Setup, Idle, Tempo sync, Time sync, Testing and Play/Dance shown in figure 17. Using the state diagram, the software architecture was design as shown in figure 18. To read in data, another software architecture was designed for this process as shown in figure 19. Our second software design which is our manual design includes 4 different states – Setup, Idle, Testing and Play/Dance shown in figure 20. Using the state diagram, the software architecture was design as shown in figure 21.

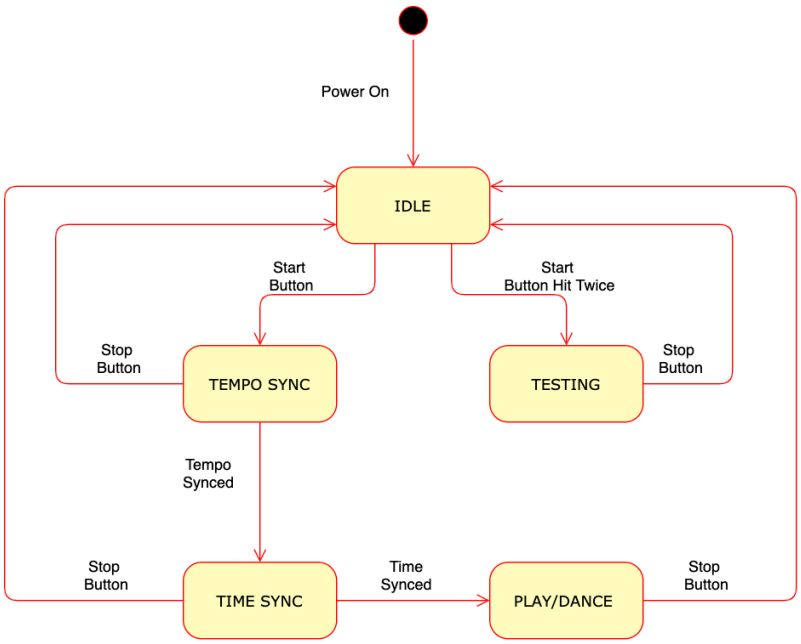


*Figure 17: State machine diagram with states idle, testing, tempo sync, time sync and play/dance*

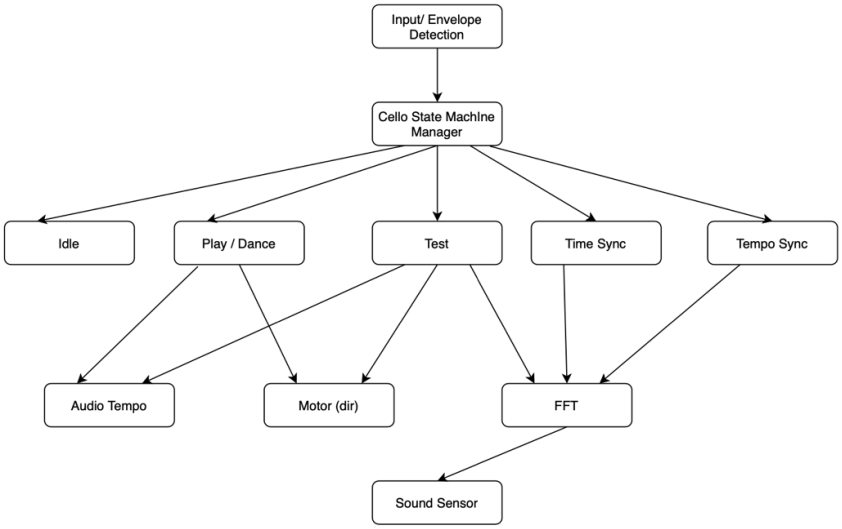


*Figure 18: Flowchart detailing the software architecture that includes the states from the state machine*

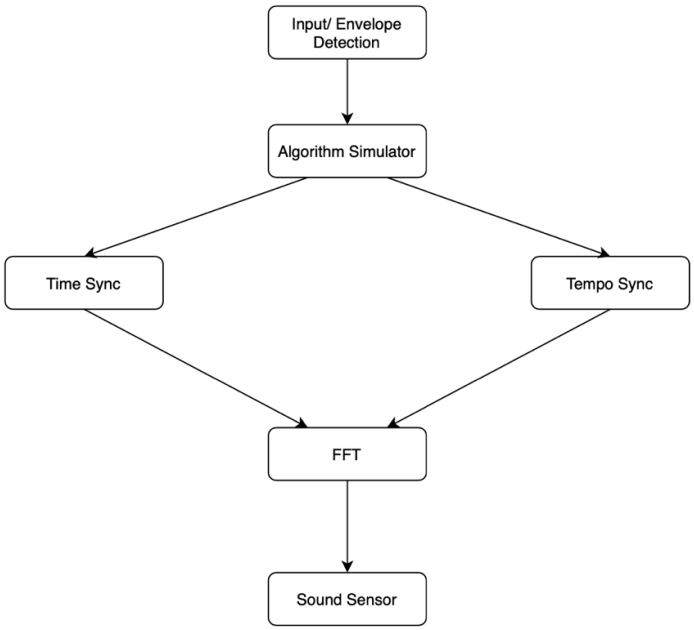


*Figure 19: Software architecture detailing the algorithm simulator*

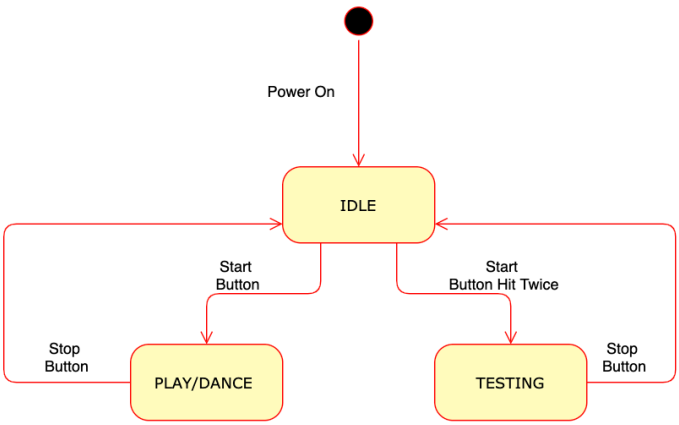


*Figure 20: State machine diagram for the manual design with states idle, testing and play/dance*

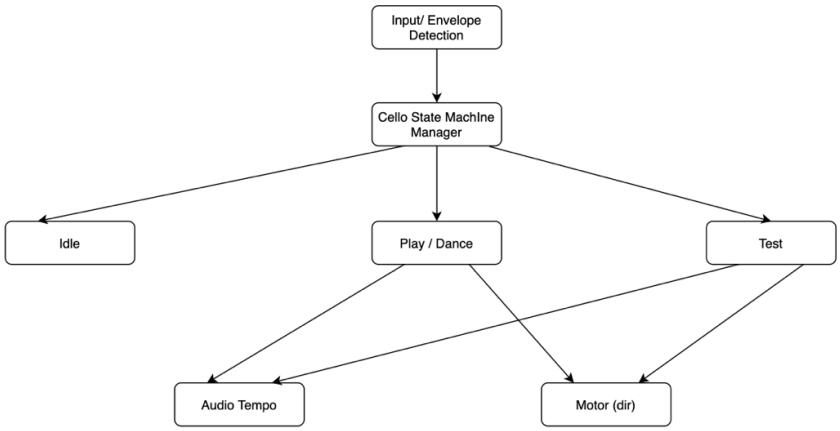


*Figure 21: Flowchart detailing the software architecture for the manual software design that includes the states from the state machine*

## Integration Plan

Each subsystem must be able to integrate and work together with each other. Various tests and simulations will be implemented to ensure they work together properly. We will design and simulate the software first, and then we can test each sub-system separately. By taking a step-by-step approach, we will be able to debug the entire system easily, rather than trying to troubleshoot the entire system when it has been fully assembled.

First implemented our software on Arduino IDE, we then integrated our design on a breadboard with the Arduino Uno microcontroller to test out the LEDs, speakers, mic, servos, the pushbuttons, and potentiometers. Then next step was to solder all the electronic components to the PCB and test it out to ensure it meet all the requirements. Lastly, we integrated our hardware and software all together on our PCB.

Software Risk Parts:

• The time sync and tempo sync implementation are both medium-risks parts of our design because the system will not be able to operate as expected. To reduce such risks, we will first design a system that works manually with the use of pushbuttons and potentiometer.

• The Arduino Uno is the most high-risk part of our design because the entire system will not function if the Arduino Uno fails. To deal with this potential issue we will test the microprocessor with each sub-system before implementing it with the whole design. By testing on a bread board, the risk of failure is reduced when integrating these parts on the PCB.

Hardware Risk Parts:

• The PCB integration is another high-risk part of the design. All the electronics connected to the PCB will need to be appropriately powered so it does not cause a blowout. To mitigate this risk, we simulated our design on a breadboard before permanently fixing components. We measured the voltage level of all the components on the PCB and compared results to its expected results. We also added two fuses for circuit protection.

Mechanical Risk Parts:

• The servo implementation of the design is high risk since we have two servos moving rotationally. To

reduce this risk, we tested the servos on our breadboard simulation using potentiometer to test the

speed of which in rotates.

## Verification Plan

*Table 2: Test matrix showing which sub-system requirements are verified by each test*

| | | Sub-System Requirement Number | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1.1 | 1.2 | 2.1 | 2.2 | 2.3 | 3.1 | 3.2 | 3.3 | 3.4 | 4.1 | 4.2 |
| Test Number | 1 | Dem | Dem | Dems | | | Dems | | | | Dems | |
| | 2 | | | | | | | | | | | |
| | 3 | | | | | | Insp | | Insp | Insp | | |
| | 4 | | | | Test | Test | | | | | | Test |
| | 5 | | | | | | | Insp | | | | Insp |

**Test 1**
This will be an external inspection of the machine to verify system meets requirements.

*Table 3: Description and procedure of sub-system verification provided by Test 1.*

| Procedure | System Requirement | Pass/Fail Criteria |
|---|---|---|
| Use measuring tape of to find the dimensions of the PCB. | Sub 3.4 | Must be within W x L: 15cm x 15cm |
| Use measuring tape to find length, width, and height of machine with power turned | Sub 3.2 | Must fit within a W x L x H: 35cm x 35 cm x 35 cm space |
| Use measuring tape to find length, width, and height of machine when power is turned on at full extension. | Sub 3.2 | Must be confined within a space of W x L x H: 75cm x 75 cm x 75 cm |
| Use scale to find the weight of the system | Sub 3.2 | Must weigh less than 20 lbs |
| Inspect that the system has two distinct motions | Sub 3.1 | Must have two separate elements each having one motor/actuator controlling movement in one spatial axis |

**Test 2**
This will be an internal inspection of the machine to verify system requirements.

*Table 4: Description and procedure of sub-system verification provided by Test 2.*

| Procedure | System Requirement | Pass/Fail Criteria |
|---|---|---|

| | | |
|---|---|---|
| Inspect that no exposed 120VAC conductors permitted anywhere in the | Sub 3.2 | No exposed 120AVC conductors |
| Inspect AC power wiring must conform to U.S. National Electrical Code (NEC) standard | Sub 4.2 | |
| Inspect that input power (AC or DC) are fused or short-circuit protected | Sub 4.2 | Must have fuses/circuit protection |
| Inspect that there is a name place on the box with voltage, current, and polarity | Sub 3.2 | |

**Test 3**
This will demonstrate the proper operation of the system.

*Table 5: Description and procedure of sub-system verification provided by Test 3.*

| Procedure | System Requirement | Pass/Fail Criteria |
|---|---|---|
| Use power switch to turn system on and machine | Sub 4.1 | Red LED will activate |
| Press “PLAY” button | Sub 1.1 | The Green LED will activate, and Cello stick will move to a specific position |
| Generates the audio sequence and moves mechanical elements in | Sub 1.2 / Sub 2.1 / Sub 3.1 | No excessive vibration or jerky motions. Smooth transitions from one |
| Musician should play some “musical instrument” in time to the music | Sub 1.1/Sub 2.2 | No excessive delays (>0.25 sec) from start of note. |
| Rotate potentiometer to control volume | Sub 1.1 | Volume speaker level will be adjusted by the user |

**Test 4**
Test the microcontroller processing and memory to verify it allows for the machine to meet system requirements.

***Table 5. Description and procedure of sub-system verification provided by Test 4.***

*Table 6: Description and procedure of sub-system verification provided by Test 4.*

| Procedure | System Requirement | Pass/Fail Criteria |
|---|---|---|
| Use power switch to turn the system on | 4.2 | Red LED will activate |
| Double tap the “PLAY” button to enter test mode | 1.2 | Both Red and Green LED will be activated to indicate it is in test mode |
| Use multimeter to probe each rail of the PCB and measure voltage | 3.2 | Voltage readings are within .2V of expected value |
| Use thermal imager to take temperature of component when on | 4.2 | No component exceeds 80 degrees Celsius |
| Operate completely in stand-alone mode, with no external PC/phone control or connections | 4.2 | Must function without external connections |
| Use oscilloscope to probe the audio output signal and record the measured value in | 2.3 | Measured values must fall between the max and min threshold values |

# Test Report

## Results for Test 4 – Test

Setup: The purpose of this test is to measure the voltage of the power supply being used, the voltage going out of the LM2596 Buck Converter and compare them with the expected values to ensure accuracy it is within to .2V of the expected value. To perform this test, test lead probes were connected in a multimeter. The multimeter was used to take the measurements, with the signal probe attached to one of the test pins to read the voltage and the other grounded with PCB board ground. A picture of the test setup is in **Figure 21**.

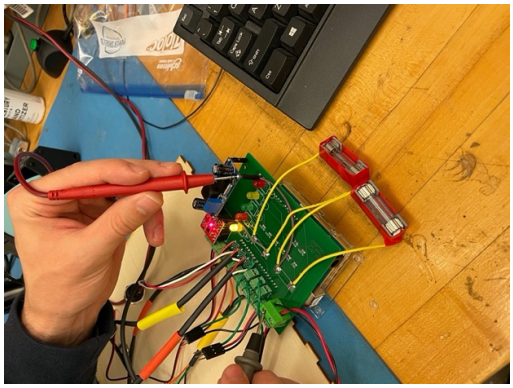

*Figure 21: A picture of the actual test setup. The PCB powered by a 9V power supply. The buck converter output is being measured.*

Measurements: To measure the voltage, the PCB was connected the 9V power supply. The on the multimeter the voltage going into the PCB was measured to make sure it was within 6V – 12V to power the Arduino uno. Then the voltage output of the block up converter was measured and adjusted to ensure it had an output voltage of 5 ± .2V.

Procedure:

1. Procedure: Connect PCB to power supply. Connect multimeter test probes to the "high" and "low" of the input of the LM2596 Buck Converter.
   Expected Output: 6V – 12V

   Requirements Tested: 8.7265V

   Results: Voltage reading was within .2V of expected value. **PASS**.

2. Procedure: Connect multimeter test probes to the "high" and "low" of the output of the LM2596 Buck Converter. Adjust the output to 5V.
   Expected Output: Output voltage is 5V

   Requirements Tested: 5.0336V

   Results: Voltage reading was within .2V of expected value. **PASS**.

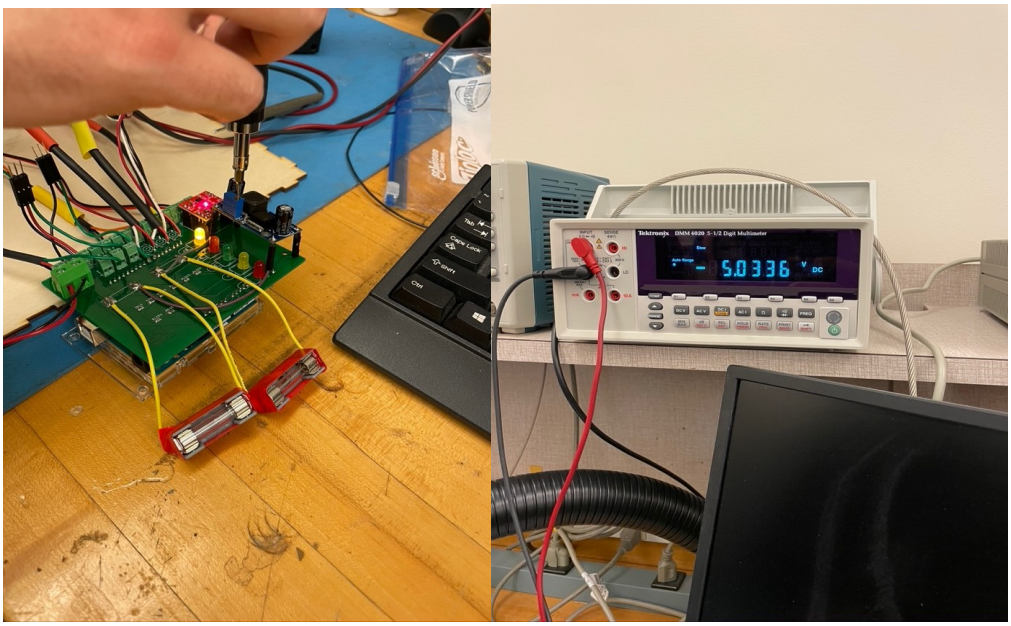


*Figure 22: A picture of the input voltage of the LM2596 Buck Converter being measured. The display of multimeter shows the voltage reading.*

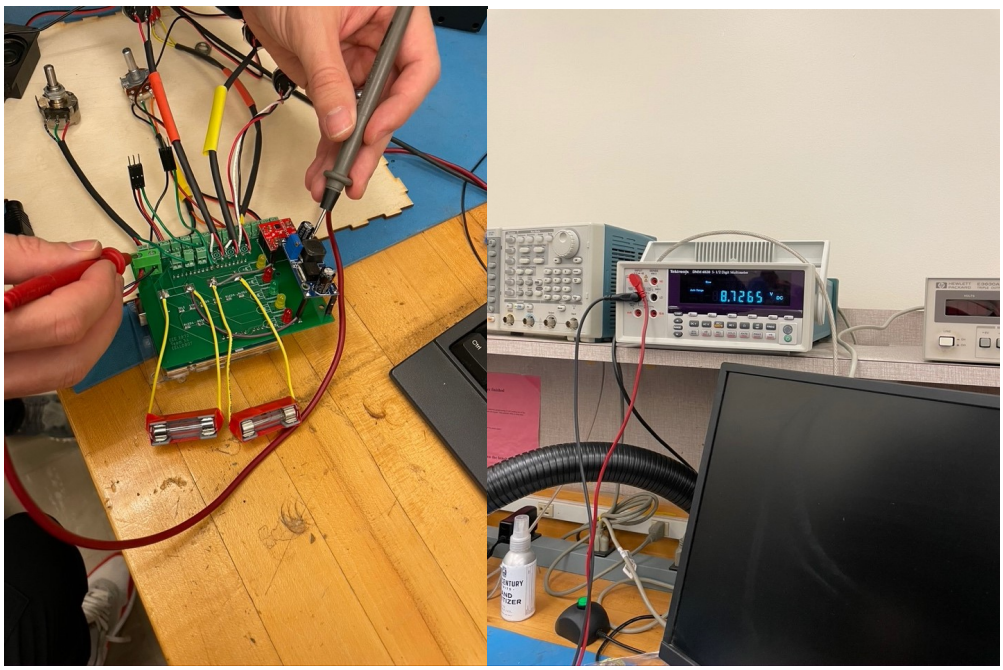


*Figure 23: A picture of the output voltage of the LM2596 Buck Converter being adjusted. The display of multimeter shows the voltage reading.*

## Conclusion

Over the course of the semester, our team was able to design and construct a functioning project with complex electrical and mechanical details. This was a rather daunting project, and our ideas changed from the beginning to the end. We took some time to realize the realities of our circumstances, and in doing so we decided to deviate from our original design and the project that resulted was successfully implemented.

Delegating tasks between team members and breaking the system into sub-systems also helped us finish this project in an efficient manner and ensured that every team member was able to contribute something unique to the overall project. Each team member was able to focus on their individual tasks without any redundancy or overlap. This helped us avoid being stretched too thin as our schedules and to-do lists grew more involved while allowing us to work in parallel and come together at the end to produce a complete product.

One thing that we could have done better from the beginning is create a more detailed weekly schedule.
Our existing schedule does not include much in terms of explicit tasks and having something more detailed would have alleviated some of the stress relating to deadlines and time constraints that we experienced. Our design notebook was also more actively updated during the former part of the semester. If we were to do this project again, we would also make the notebook more detailed in order to help us keep track of the path of ideas and plans that we had. We also faced roadblocks along the way that caused us to halt our linear progression. One such challenge was getting the software to function properly 100 percent of the time without significant user interaction and manual manipulation needing to be applied. Despite these challenges, we were able to produce a finished product of which we are very proud.

Overall, we are proud of the progress we were able to make as a team, and our project turned out to be a successful endeavor. We faced challenges along the way, but we were eventually able to end the project with positive attitudes.

| **Role** | **Leader** |
|---|---|
| Team Lead | Aaron Hoffman |
| Power sub-system | Kennedy Houston |
| Mechanic sub-system | Kennedy Houston |
| Audio sub-system | Aaron Hoffman |
| Controller sub-system | Andre Mbakwe |

# Appendix: SW

SOURCE CODE FOR THE CELLOBOT SIMULATION WITH VARIOUS USER INPUT DEVICES

```
#include <Servo.h>
#include <stdio.h>
#include <math.h>
#define TempoCal 512
#define TempoPotMax 1023
#define PwmMax 255
#define TempoResolution 0.5
#define rest 0
#define speakerPin 6
#define baseServoPin 7
#define armServoPin 8
#define ANALOG_PIN A1
#define MIC_PIN A2
#define TONE_PIN 2
#define NUM_STATES 3

int Octive = 2;
//Music Notes based on Octive--
double C,D,E,F,G,A,B, high_C;
// the pin that the pushbutton is attached to
const int  Start = 4;
const int  Stop = 3;
// the pin that the LED is attached to
const int idlePin = 13;

const int testPin = 11;
const int playPin = 9;
```

```
unsigned long timePress = 0;

unsigned long timePressLimit = 0;

int clicks = 0;

int song_tempo = 250;

int baseIncrement = 1;

int armIncrement = 0;

int basePos = 0;

int armPos = 90;

//setup the servo output

Servo baseServo;

Servo armServo;

/*****************************************************************

  Function Prototypes

*****************************************************************/

void IDLE_STATE(void);    // State IDLE

// void TEMPO_STATE(void);   // State TEMPO

void TEST_STATE(void);    // State TEST

// void TIME_STATE(void);    // State TIME

void PLAY_STATE(void);    // State PLAY


/*****************************************************************

  State Machine Skeleton

*****************************************************************/

 // enum of each state

 typedef enum

 {

     STATE_ONE,
```

```
    // STATE_TWO,
    STATE_THREE,
    // STATE_FOUR,
    STATE_FIVE
 }StateType;
 // define state machine table structure
 typedef struct
 {
    StateType State;       // Define the command
    void(*func)(void);     // Defines the function to run
 }StateMachineType;
// Table of valid states of the sm and function to be executed for
StateMachineType StateMachine[] =
 {
    {STATE_ONE, IDLE_STATE},
    // {STATE_TWO, TEMPO_STATE}
    {STATE_THREE, TEST_STATE},
    //{STATE_FOUR, TIME_STATE}
    {STATE_FIVE, PLAY_STATE}
 };
// Store current state of state machine
StateType SM_STATE = STATE_ONE;
/*****************************************************************
  Initialization
*****************************************************************/
void setup()
{
  pinMode(Start, INPUT);
```

```
  pinMode(Stop, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(Stop), IDLE_STATE, CHANGE);
  pinMode(ANALOG_PIN, INPUT);
  pinMode(MIC_PIN, INPUT);
  pinMode(TONE_PIN, INPUT);
  Serial.begin(9600);
  //set up outputs
  pinMode(idlePin, OUTPUT);
  pinMode(testPin, OUTPUT);
  pinMode(playPin, OUTPUT);
  pinMode(speakerPin, OUTPUT);
  baseServo.attach(baseServoPin);
  armServo.attach(armServoPin);
  //set the servo to zero initial condition.
  baseServo.write(0);
  armServo.write(80);
}
/*********************************************************************
  Main Loop
*********************************************************************/
void loop()
{
  // Start the state machine
  RUN_STATEMACHINE();
}

/*********************************************************************
```

```
  IDLE STATE:
  • Reset memory
  • Hit start to begin tempo sync
  • Double tap start to go into the TEST state
*****************************************************************/
void IDLE_STATE(void)
{
    digitalWrite(idlePin, HIGH);
    noTone(speakerPin);
    baseServo.write(basePos);
    armServo.write(armPos);
    /*if (digitalRead(tempoPin) == HIGH) {

      digitalWrite(tempoPin, LOW);
      SM_STATE = STATE_ONE;

    }
    */
    if (digitalRead(testPin) == HIGH){
      digitalWrite(testPin, LOW);
      SM_STATE = STATE_ONE;

    }
    /*else if (digitalRead(timePin) == HIGH) {
      digitalWrite(timePin, LOW);
      SM_STATE = STATE_ONE;
    }*/
    else if (digitalRead(playPin) == HIGH) {
```

```
  digitalWrite(playPin, LOW);

  SM_STATE = STATE_ONE;

}
if ( digitalRead(Start) == HIGH) {

  delay(200);

  if (clicks ==0) {

    timePress = millis();

    timePressLimit = timePress + 1000;

    clicks = 1;

   }

  else if ( clicks == 1 && millis() < timePressLimit) {

    // Button pressed twice

    //set variables back to 0

    timePress = 0;

    timePressLimit = 0;

    clicks = 0;
     digitalWrite(idlePin, LOW);

    SM_STATE = STATE_THREE;

  }

}

  if (clicks == 1 && timePressLimit != 0 && millis() > timePressLimit){

    // Pressed once

    timePress = 0;

    timePressLimit = 0;

    clicks = 0;

    digitalWrite(idlePin, LOW);

    SM_STATE = STATE_FIVE;

   }
```

```
}



/*****************************************************************
  TEMPO STATE:
  • Write function to read in audio and determine octive level tempo
  • Go to IDLE when stop is hit

 Write code to determine tempo sync

*****************************************************************/
/* void TEMPO_STATE(void)
{
    digitalWrite(tempoPin, HIGH);

    if (digitalRead(Sync) == HIGH){
      digitalWrite(tempoPin, LOW);
      delay(1000);
      SM_STATE = STATE_FOUR;
    }
}
*/
/*****************************************************************
  TEST STATE:
  • PLay default audio sound
```

```
  • Make Robot move
  • Go to IDLE when stop is hit

  write code for robot movement and audio out
*****************************************************************/
void TEST_STATE(void)
{
    digitalWrite(testPin, HIGH);
    int duration;
    int tempo;
    int tempo_pot; // default
    int toneSelector;
    int speed;

    //play the song
    int i_note_index = 0;

    while (digitalRead(testPin))
    {

      // Change octive
      if (digitalRead(TONE_PIN)  == HIGH){
        Octive ++;
      }
      if ((Octive > 9) && (digitalRead(TONE_PIN) == HIGH)){
        Octive = 2;
      }
```

```
      //Music Notes based on Octive--

      C = 16.3516*pow(2,Octive);

      D = 18.35405*pow(2,Octive);

      E = 20.60172*pow(2,Octive);

      F = 21.82676*pow(2,Octive);

      G = 24.49971*pow(2,Octive);

      A = 27.5*pow(2,Octive);

      B = 30.86771*pow(2,Octive);

      high_C = 32.70320*pow(2,Octive);


      //Row Row Row Your Boat

      int songLength = 54;

      double notes[] = {C, rest, C, rest, C, rest, D, rest, E, rest, E, rest,
D, rest, E, rest, F, rest, G, rest, high_C, rest, high_C, rest, high_C, rest,
G, rest, G, rest, G, rest, E, rest, E, rest, E, rest, C, rest, C, rest, C,
rest, G, rest, F, rest, E, rest, D, rest, C, rest};

      int beats[] =
{2,1,2,1,2,1,1,1,2,1,2,1,1,1,2,1,1,1,6,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,2,1,1,1,2,1,1,1,5,1};


      tempo_pot = analogRead(ANALOG_PIN);

      tempo = song_tempo*float(tempo_pot)/TempoCal; //read the tempo POT




      //set the Servo speed

      speed = (1 - float(tempo_pot)/TempoPotMax)*PwmMax;


      // Base servo movement
```

```
if (basePos >= 180)
{
  baseIncrement = 0;
}
else if ( basePos <= 0){
  baseIncrement = 1;
}

if ( baseIncrement == 1)
{
  basePos ++;
}
else
{
  basePos --;
}

baseServo.write(basePos);
delay(speed / 10);

// arm servo movement
if (armPos <= 0)
{
  armIncrement = 1;
}
else if ( armPos >= 180){
  armIncrement = 0;
```

```
    }


    if ( armIncrement == 0)
    {
      armPos --;
    }
    else
    {
      armPos ++;
    }
    armServo.write(armPos);
    delay(speed / 10);

    // Play song
    duration = beats[i_note_index] * tempo;
    tone(speakerPin, notes[i_note_index], duration);
    delay(duration);

    //increment the note counter
    ++i_note_index;
    if(i_note_index >= songLength)
    {
      i_note_index = 0;
    }

  }
}
```

```
/*****************************************************************
  TEMPO STATE:
   • Write function to read in audio and sync the time
   • Go to IDLE when stop is hit

  Write code to determine time sync
*****************************************************************/
/*void TIME_STATE(void)
{
    digitalWrite(timePin, HIGH);
    if (digitalRead(Sync) == HIGH){
       digitalWrite(timePin, LOW);
       SM_STATE = STATE_FIVE;
    }
}
*/
/*****************************************************************
  PLAY STATE:
  • Audio out and robot dance for certain amount of time

  Write code to play audio and move robot
*****************************************************************/
void PLAY_STATE(void)
{

    digitalWrite(playPin, HIGH);
```

```
int duration;
int tempo;
int tempo_pot; // default
int toneSelector;

int speed;

//play the song
int i_note_index = 0;



while (digitalRead(playPin))
{
  // Change octive
  if (digitalRead(TONE_PIN)  == HIGH){
    Octive ++;
  }
  if ((Octive > 9) && (digitalRead(TONE_PIN) == HIGH)){
    Octive = 2;
  }




  //Music Notes based on Octive--
  C = 16.3516*pow(2,Octive);
  D = 18.35405*pow(2,Octive);
  E = 20.60172*pow(2,Octive);
```

```
      F = 21.82676*pow(2,Octive);

      G = 24.49971*pow(2,Octive);

      A = 27.5*pow(2,Octive);

      B = 30.86771*pow(2,Octive);

      high_C = 32.70320*pow(2,Octive);


      //Row Row Row Your Boat

      int songLength = 54;

      double notes[] = {C, rest, C, rest, C, rest, D, rest, E, rest, E, rest,
D, rest, E, rest, F, rest, G, rest, high_C, rest, high_C, rest, high_C, rest,
G, rest, G, rest, G, rest, E, rest, E, rest, E, rest, C, rest, C, rest, C,
rest, G, rest, F, rest, E, rest, D, rest, C, rest};

      int beats[] =
{2,1,2,1,2,1,1,1,2,1,2,1,1,1,2,1,1,1,6,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,2,1,1,1,2,1,1,1,5,1};


      tempo_pot = analogRead(ANALOG_PIN);

      tempo = song_tempo*float(tempo_pot)/TempoCal; //read the tempo POT




      //set the Servo speed

      speed = (1 - float(tempo_pot)/TempoPotMax)*PwmMax;


      // Base servo movement

      if (basePos >= 180)

      {

        baseIncrement = 0;

      }

      else if ( basePos <= 0){

        baseIncrement = 1;
```

```
}

if ( baseIncrement == 1)
{
  basePos ++;
}
else
{
  basePos --;
}

baseServo.write(basePos);
delay(speed / 10);

// arm servo movement
if (armPos <= 0)
{
  armIncrement = 1;
}
else if ( armPos >= 180){
  armIncrement = 0;
}

if ( armIncrement == 0)
{
  armPos --;
}
```

```
      else
      {
        armPos ++;
      }
      armServo.write(armPos);
      delay(speed / 10);

      // Play song
      duration = beats[i_note_index] * tempo;
      tone(speakerPin, notes[i_note_index], duration);
      delay(duration);

      //increment the note counter
      ++i_note_index;
      if(i_note_index >= songLength)
      {
        i_note_index = 0;
      }

    }
}

/*****************************************************************
  Run State Machine
*****************************************************************/
void RUN_STATEMACHINE(void)
{
```

```
    // Make Sure States is valid
    if (SM_STATE < NUM_STATES)
    {
        // Call function for state
        (*StateMachine[SM_STATE].func)();
    }
    else
    {
        // Code should never reach here
        while(1)
        {
            // Some exception handling.
        }

    }

}
```