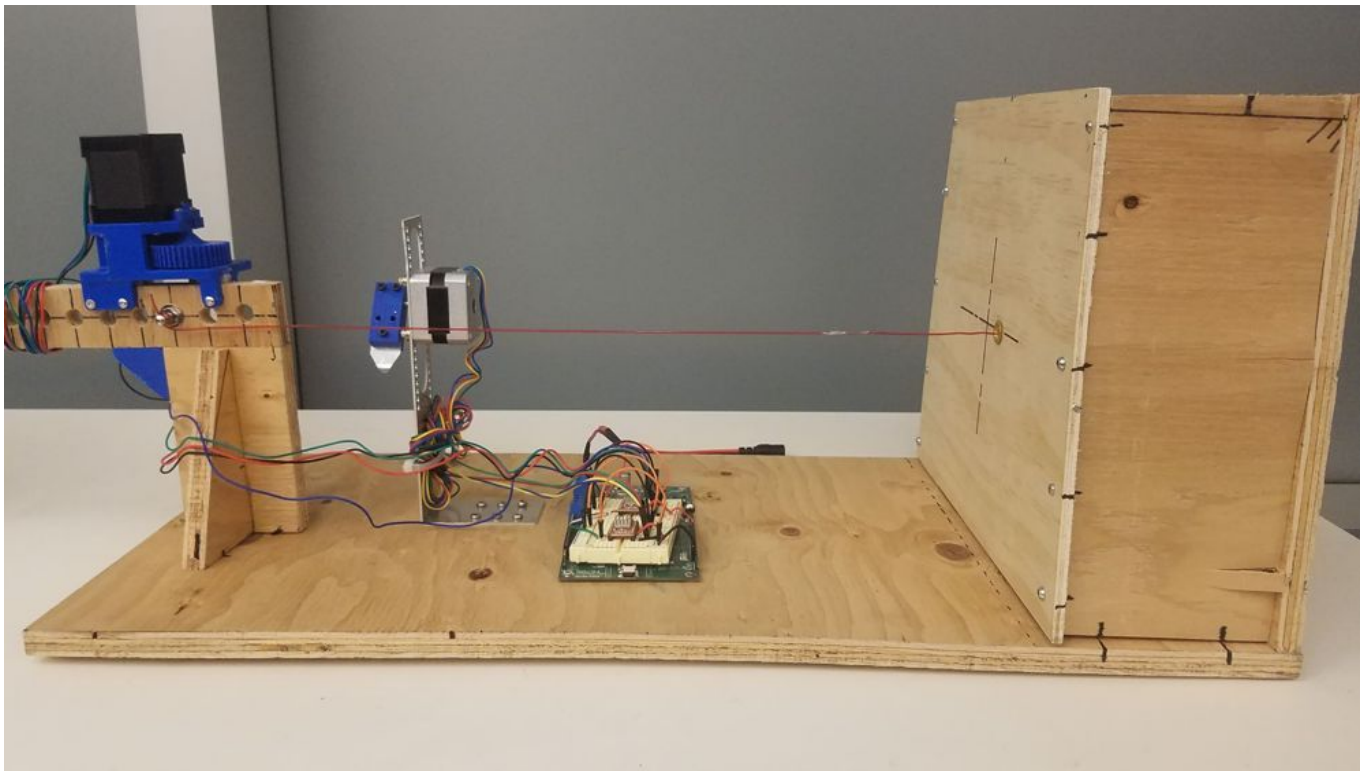
MECH 423 FINAL PROJECT REPORT: SINGLE STRING MUSICAL INSTRUMENT

GROUP MEMBERS:

JUAN BUSTAMANTE, 22637145, j-e-bustamante@hotmail.com

RISHABH SINGAL, 18802141, rishabhsingal01@gmail.com

DATE OF SUBMISSION: DECEMBER 10, 2018



1. Abstract

The goal of this project is to design an innovative single string musical instrument based on the concept of changing string tension to change the natural frequency of vibration of the string. The instrument resembles guitar but only uses one string to play notes over multiple octaves.

A soundboard is designed to amplify the natural frequency of the vibrating string by vibrating wood and air at the same frequency. Furthermore, the string is offset from the center of the soundboard which prevents destructive interference of sound waves.

One stepper motor varies the tension of the string by turning a tuning knob using a custom, 3D printed gear reduction set. The tuning knob has a high gear reduction because it uses a spur-worm gear set which prevents the tuning knob from being back-driven and allows relatively small stepper motor to deliver enough torque. The other motor plucks the string after the desired tension has been achieved. The plucker is made out of a 0.45mm thick aluminum sheet which provides it flexibility when it hits the string.

The motors are controlled using Pololu stepper motor drivers which receive stepping signals from the MSP board. The MSP board commands the tuning motor's motor driver first to go to a specified position followed by commanding the plucking motor's motor driver. The synchronization of these two motors enables us to play songs using the device.

The software interface allows the user to play piano notes over one octave and jingle bells. The user receives feedback from the interface as the pressed piano key lights up for a pre-defined amount of time. Furthermore, the user has the ability pause, play and/or stop the *Jingle Bells* song while the song sequence is in progress. This is achieved by using task feature in C#.

2. Objectives

The overall goal of the project was to develop an innovative musical instrument which only uses one harp string to produce notes over 3 octaves. Our plan was to design a quick response system which can reproduce a hard-coded song and play it with a small delay such that the song is perceptible to the user.

Since no one has designed a single string musical instrument before, we had to base the soundboard dimensions off of the harp soundboard dimensions and limit them such that the device can be ergonomically carried around. The final sound produced by the

plywood soundboard was surprisingly very pleasing to listen to given that procuring proper citka spruce soundboard material in time would have been extremely expensive and difficult. Furthermore, the tuning knob in combination with the custom 3D printed 4:1 gear reduction set worked really well as it was able to provide a much higher than needed level of tension to the string. Using the notes that we identified in octave 3, we were able to successfully implement a piano feature as well as play *Jingle Bells* on the device.

After designing the plucker, we observed that the plucker caught the string at low-frequency notes as the stepper motor did not provide enough torque to overcome the counteracting force from the string. Due to time constraints, we focussed our efforts on identifying the high-frequency notes on which the plucker did not have any problem hitting the string hard and not catching it. But this limited the range of notes that we identified to 1 octave even though the device is still capable of producing notes in 3 octaves. Had we used DC motors instead of stepper motors for both tuning and plucking, we would have obtained higher torque and much higher response times but the project's complexity would have increased by at least an order of magnitude. There was a risk of running out of time had we decided to use DC motors but the final product would've likely been of higher quality.

3. Rationale

The idea for this project started from the concept of our Capstone project, where we need to tune the natural frequency of a vibrating mast and match that of the incoming wind. The idea of tuning natural frequency quickly became an instrument due to our interest in music. After a trip to a music store, we discovered that the best music source to attempt to tune its natural frequency is a string, since they are made for reproducing music and can be found inside many instruments. To tune the natural frequency of a string you can either change the length, mass, or tension. We found varying the tension would probably be the easiest way. We wanted to pursue this project not only due to our heavy interest in music, but also because it provided us an excellent opportunity to learn about position control systems, since we required very precise movements from our stepper motors.

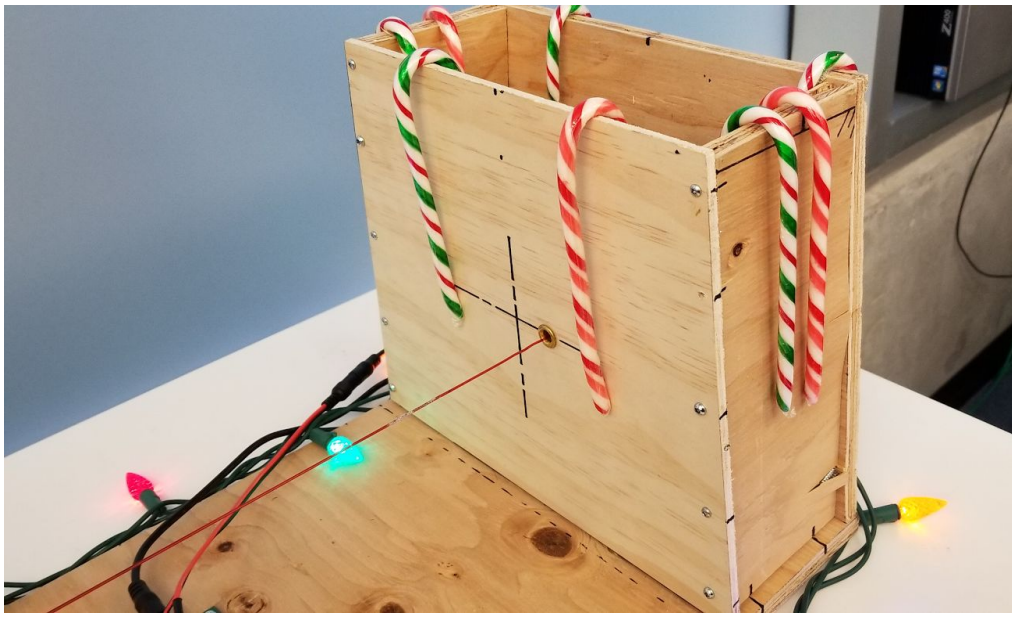
4. Summary of Functional Requirements

Functional Requirements	% Effort	Responsible Person
FR#1: Soundboard Design and Mechanical Assembly	25	Juan and Rishabh
FR#2: Motor Attachments	30	Juan (80%) and Rishabh (20%)
FR#3: Plucking + Tuning MSP430 Code	15	Juan and Rishabh
FR#4: Piano Playing C# Code	15	Rishabh
FR#5: Song Playing MSP430 Code	15	Juan and Rishabh

Description:

1. **Soundboard design and mechanical assembly:** Wooden soundboard design for amplification of harp string vibration and mechanical assembly of soundboard, harp string and base
2. **Motor Attachments:** 3D printed 4:1 gear reduction mechanism for stepper motor to turn tuning knob for tensioning the string; sheet metal L bracket for another stepper motor to pluck the string
3. **Plucking + Tuning MSP430 code:** MSP C code with timer and UART interrupts to receive commands from C# and send pulses to A4988 stepper motor drivers through digital output pins
4. **Piano playing C# code:** Intuitive C# interface for users to play piano notes over one octave which would be replicated by the musical instrument
5. **Song playing MSP430 code:** C# interface for users to play *Jingle Bells* song with an ability to play, pause and stop the song while it is in progress

5. Functional Requirement #1: Soundboard Design and Mechanical Assembly



A. Approach and Design

This requirement encompasses the design and assembly of soundboard with the base of the instrument and the harp string. It can be broken down in the following:

1. Sound Board
   The purpose of the soundboard is to amplify the natural frequency of the vibrating string to make the musical notes audible.
   - The front of soundboard is trapezoidal to resemble that of the harp and is made of ¼” thick plywood to to ensure mechanical vibration when string is plucked.
   - The sides and back of the soundboard are made of ½” thick citka spruce plywood to provide structural strength and stability.
   - The harp string is supported at the back of the front plywood using a wooden dowel. The brass ring prevents the string from rubbing against the wood and breaking.
   - The harp string entry location into the soundboard is offset by 25mm from the center to prevent sound waves from destructively interfering with each other.

2. Harp String
   The harp string present in the device is a middle C musical note and is 0.5m long. The tension of the string is modified using the tuning knob (controlled using a stepper motor) which changes the natural frequency of the string and produces a different musical note when plucked. The range of musical notes extends to more than 2 octaves which can be used to play various songs.

B. Inputs and Outputs

The input to this system is the plucking of string using a plucking motor which induces vibrates the string, soundboard wood and soundboard air at the string's natural frequency. The output is the musical note (which sounds like a guitar) produced due to induced vibrations.

C. Parameters

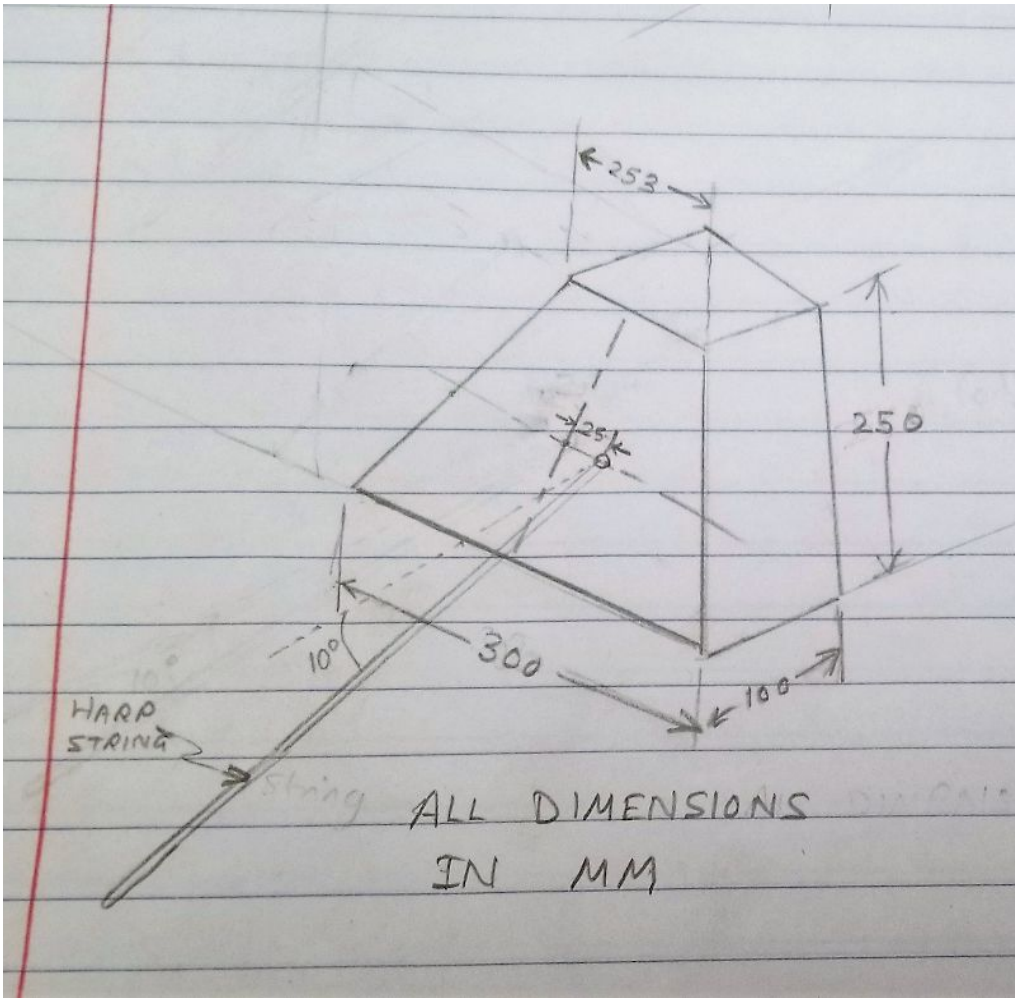


Figure: *Soundboard dimensions in single string musical instrument*

The parameters of the system are as follows:

1. **Dimensions of the soundboard:** Since a single string musical instrument has never been designed before, the soundboard dimensions are based on the soundboard dimensions of a harp.

   The research conducted on harp soundboard vibrational characteristics by Waltham et. al (2008) mentioned the following values of dimensions of a soundboard:

Table 1: *Typical soundboard dimensions of harp instrument (Waltham et. al, 2008)*

h	1.4	m
lmax	0.5	m
lmin	0.1	m
dmin	2.25	mm
dmax	11.5	mm

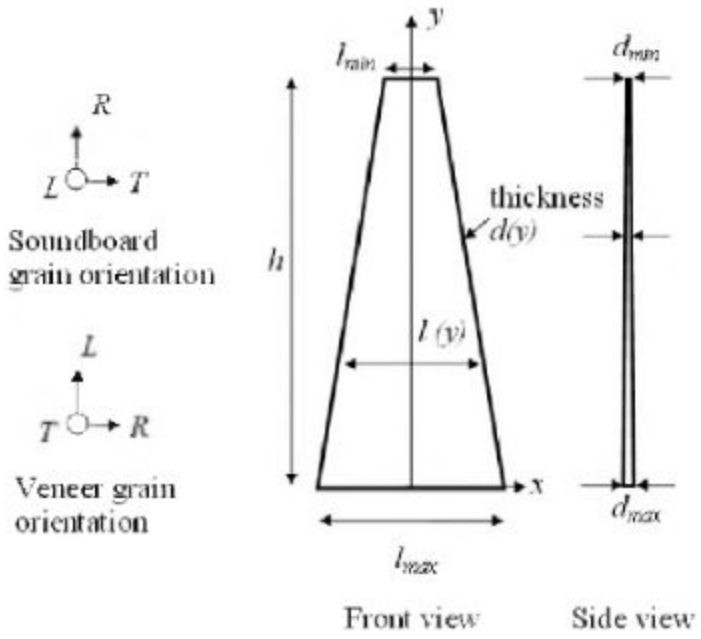


Fig. 2. Layout of harp soundboard, showing the x-y coordinate system used in this paper, and the orientation of the wood grain for the soundboard base and veneer. The longitudinal (L), transverse (T) and radial (R) directions refer to the natural cylindrical coordinate system of a tree limb.

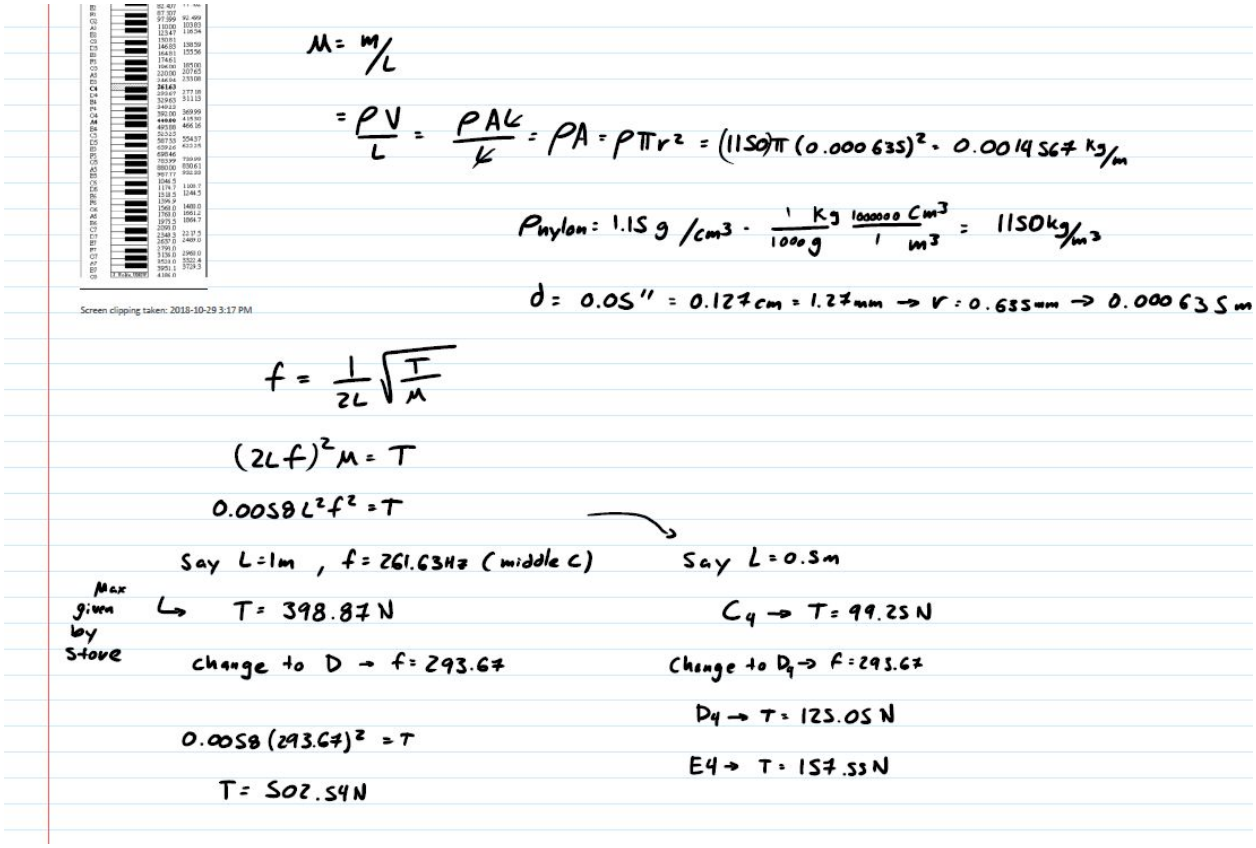
Figure: *Harp soundboard dimensions (Waltham et. al, 2008)*

In our instrument, the l_{max} is set to 300mm instead of 500mm so that the instrument can fit inside the MECH locker. The l_{min} is set at 235mm so that it corresponds to note C5 on a typical harp. The height is set to 250mm and is restricted by the amount of citka spruce wooden plywood material that was available.

Finally, the harp string entry position is offset from the center by 25mm to ensure that the sound waves produced due to vibration of air do not destructively interfere with each other.

2. **Harp String Dimension:**
   The harp string purchased from Long and McQuade is 0.050” in diameter and 1m long. It is cu to 0.5m length because this length allows a wide range of musical notes without the risk of string over-tensioning and breaking. Since frequency is inversely proportional to the length in f = 1/(2L) * sqrt(T/M) equation, the smaller the length, the higher the frequency for the given tension and mass density.

   The maximum allowable tension on the string is 500N (obtained from Long and McQuade staff).



D. Testing and Results

Originally, the front face of the soundboard was made of ½” citka spruce plywood. However, that turned out to be very thick and produced very high decay sound (i.e. the note faded away very quickly after being produced). Hence, two more soundboards were constructed out of 1/16” aluminum sheet and ¼” plywood. Both seemed like promising options and plywood face was attached to the soundboard before aluminum. Since the plywood produced very decent quality sound, due to time constraints, the aluminum sheet was not tested with the rest of the soundboard.

Functional Requirement #2: Motor Attachments

A. Approach and Design

The objective of this functional requirement is to create custom components that can enable the two stepper motors in our system achieve their respective objectives. The components were 3D printed using the ABS printers in the MECH Machine Shop.

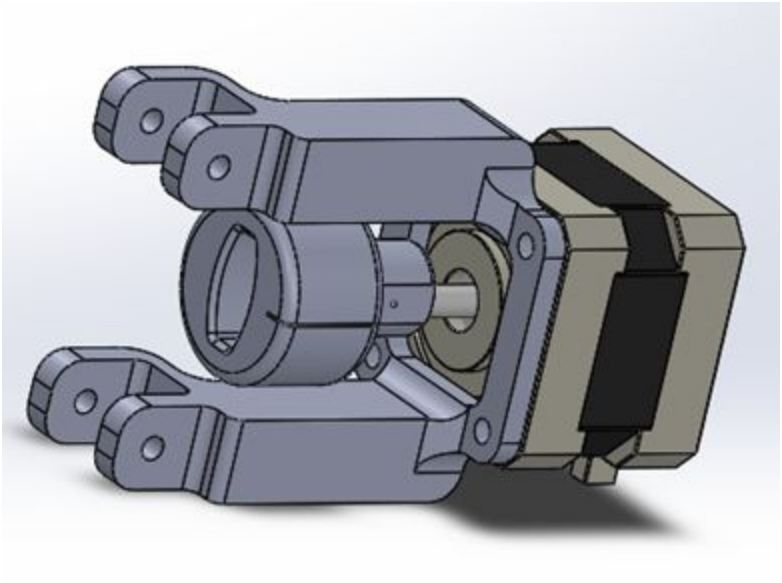
Tuning Motor:

Creating the attachments to have the tuning motor achieve its objective was a lot harder than expected. The objective of the tuning motor is to rotate the knob in the guitar string tuner shown in the figure below to change the tension, and therefore the natural frequency, of the string. Each note would be related to a number of steps from a reference position of the stepper motor. This reference position would be assigned to an specific note and then every other note we want to reproduce would be displaced from that reference. Once the program resets, the board thinks that the motor is sitting at its reference position. We have to make sure to always return the stepper motor to its reference position before turning the device off to avoid any issues with the following notes.

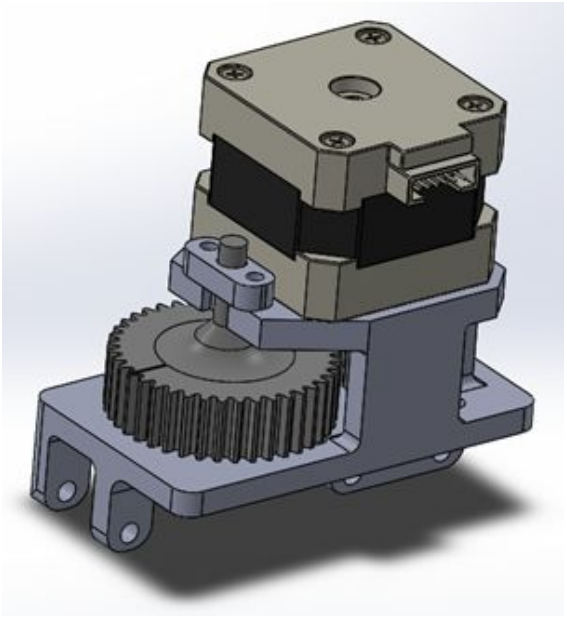
The first approach was to have a direct connection with the knob and the motors shaft. This design was ideal because stepper motors are known to have low speeds, and we

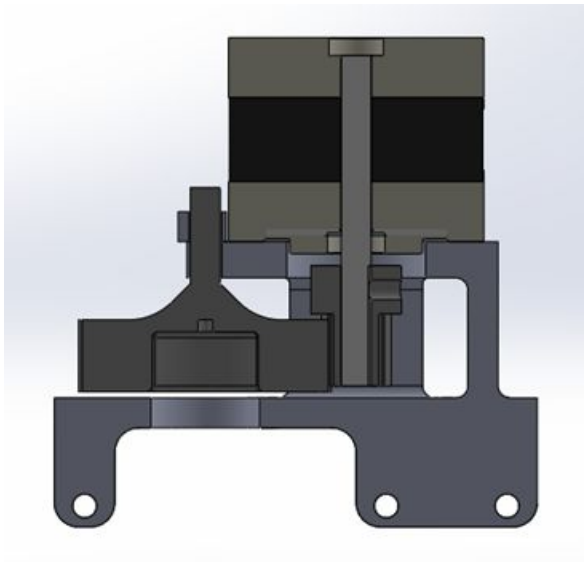
require notes to be switched fast to make the music as similar to its original version. The biggest unknown was if the stepper motor had enough torque to turn the guitar knob. We acquired the guitar knobs from amazon first. These knobs work by having a worm drive like the one shown in the figure below. This reduces the torque required to tension the string, but also prevents the gear from moving when the screw is not moving. This feature is very useful for our project because we don't want to have the stepper motor fight the tension in the string at all times, only when it is being changed.

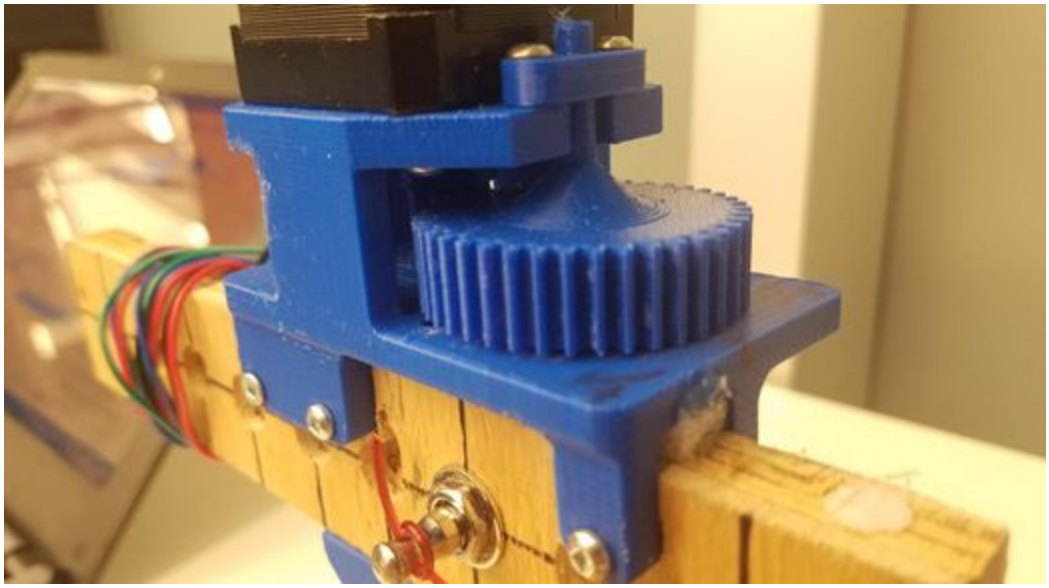
Once we received the knobs, it was hard to tell if the motor would have enough torque to turn it. The first stepper motor we used had 0.25Nm stall torque, which we did not know if it was enough. We decided the fastest way to test if the motor was capable was to 3D print a shaft attachment that would turn the knob. The shaft attachment in the figure below was modelled using SolidWorks, by measuring the knob and the shaft using calipers.



Once the print was done, we quickly tested if the motor was capable, and it was able to turn the knob with no problem, but once the harp string was attached to the knob and there was even the slightest tension on the string, the motor started skipping steps. Our next approach was to try out a bigger motor with double the stall torque: 0.6Nm. We only needed to reach an acceptable range of tension which could give us a one octave of notes. Trying the bigger motor with the same direct motor attachment was also a failed result. It was able to tension the string more than the other one, but not enough to be in our acceptable range. Our TA, Sam Berryman, then suggested to use a gearbox to solve this issue. The gearbox shown in the figures below was designed referenced of a model from GrabCad user Sam Lihn. The model found was a 4:1 gearbox for a extruder head using the same stepper motor. We modified the gears to fit the guitar knob inside and created a custom chassis to rigidly hold the motor and gears in place with the wooden frame. The quality of the 3D printed gears was surprisingly good, and they meshed easily. The gearbox was assembled in the device and now the motor was able to turn the knob easily with high tension, reproducing high pitch notes. This method sacrificed speed between notes, but we were very happy with it anyways.



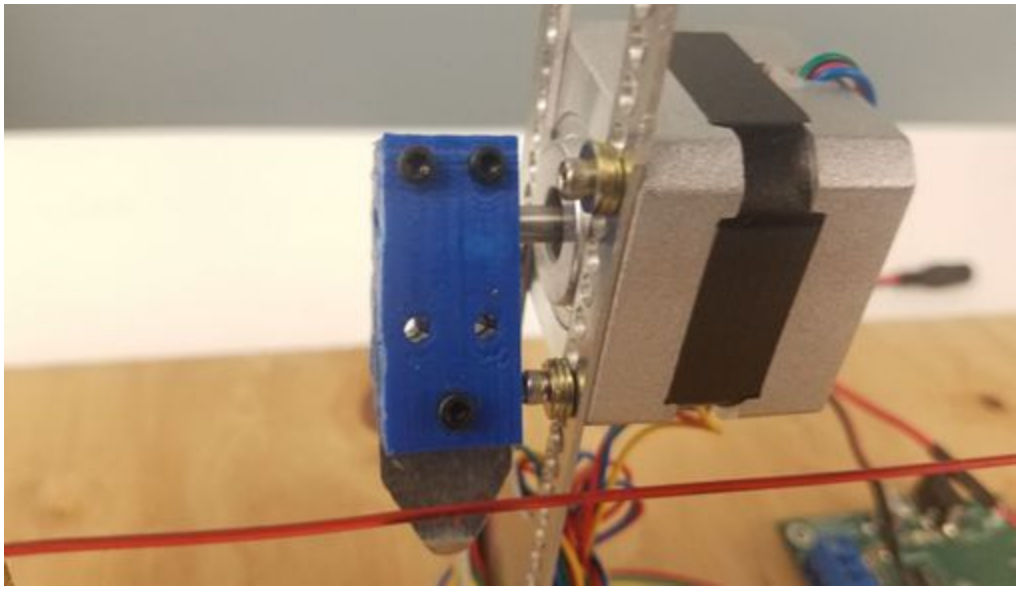


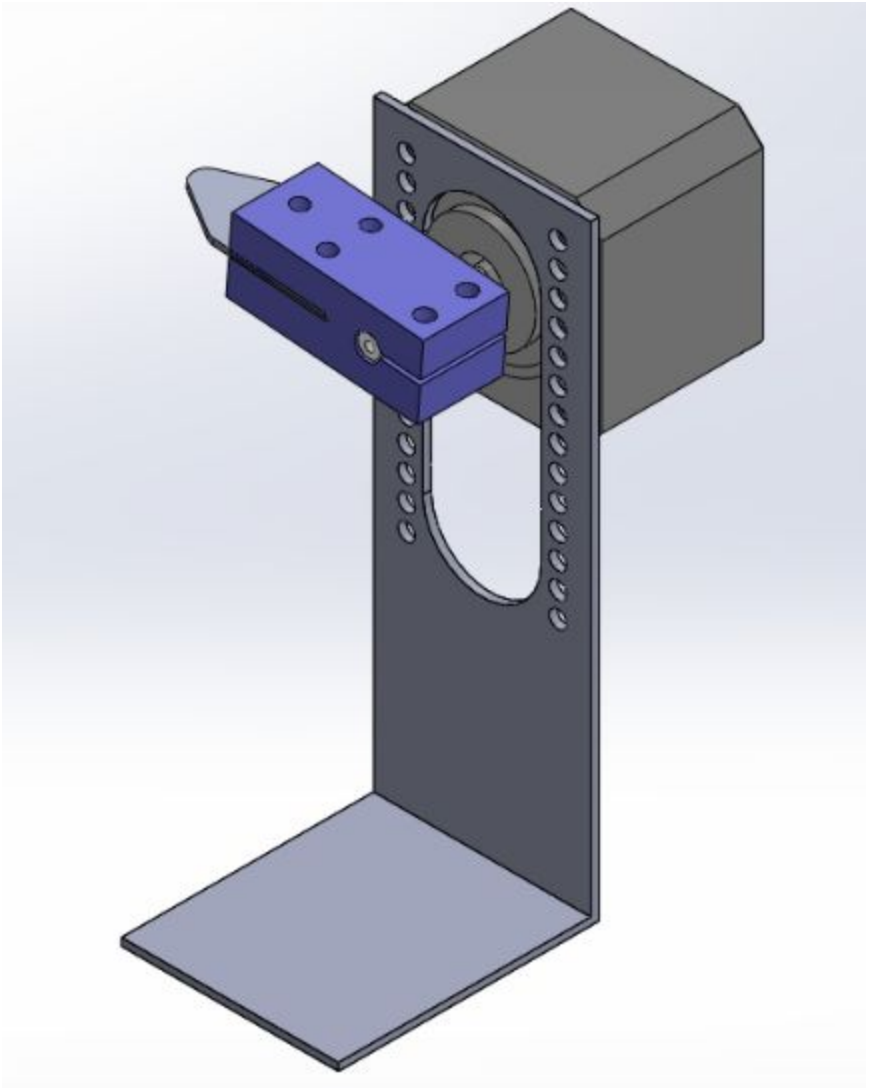


Plucking Motor:

Designing the attachments for the plucking motor were not as critical as the tuning motor, since we were able to test the notes from the tuning motor using our hands. The shaft attachment designed for the plucking motor was a two-clamp system which clamped to the shaft to prevent rotation and clamped to a small sheet metal piece. This piece is simulating a guitar pick to pluck the string. The motor was held on a bent waterjet piece that hat multiple positioning holes. The metal pick at the end of the shaft attachment could also move inwards and outwards. These two adjustable dimensions allowed us to play with the positioning of the plucking relative to the string. We quickly found that the metal piece would get stuck in the string, specially at low tensions.

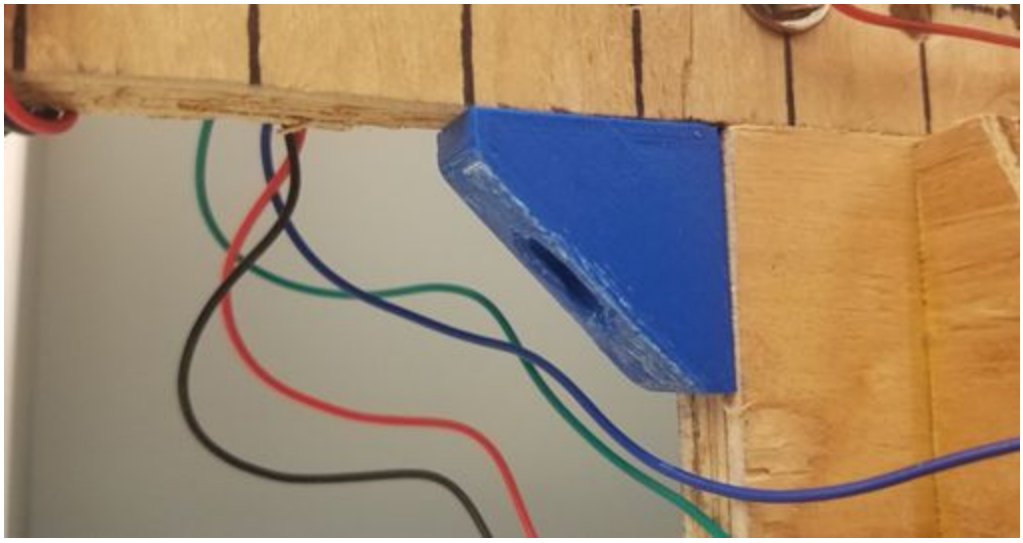
Bending the tip of the metal pick prevented this from happening. There was a clear trade-off between reproducing a good noise at high vs low tensions. These dimensions were fiddled for a while before we found an optimal position which we were happy with the sound it made, prioritizing high tensions. We then calibrated our octave around these tensions.





Supports:

Some pieces from the wooden assembly were held only using wood glue. This connection specifically had to support the full tension of the string using only the shearing holding force of the wood glue. The way it was assembled did not allow for any wood screws to support this connection, so a 3D printed support was designed to help the wood glue support the tension. It is unknown if it would have worked without it, since we decided not to test it before having this extra support.



B. Inputs & Outputs

Inputs:

Motor Torque:

The main input to this functional requirement is the torque input from both stepper motors. As mentioned before, torque was a big issue for this system because the torque required to turn the guitar tuning knob at tension was a lot higher than we expected, and way beyond the capabilities of our motors. Once the gearbox was included, the maximum stall torque was increased to 2.36Nm, which was enough to reproduce our notes.

Motor Reference Position:

Another very important input was the position which the motor started when the program was run. On every instance that the program ran, it always assumed the tuning motor was at the reference position. It was very important that we disconnected the device with the motor at its reference position, and this sometimes caused issues. When it happened that the program ran with the motor in a different position, the string would go higher than the tension we desired, and we would have to quickly disconnect the power to prevent the string from snapping.

Outputs:

Note frequency:

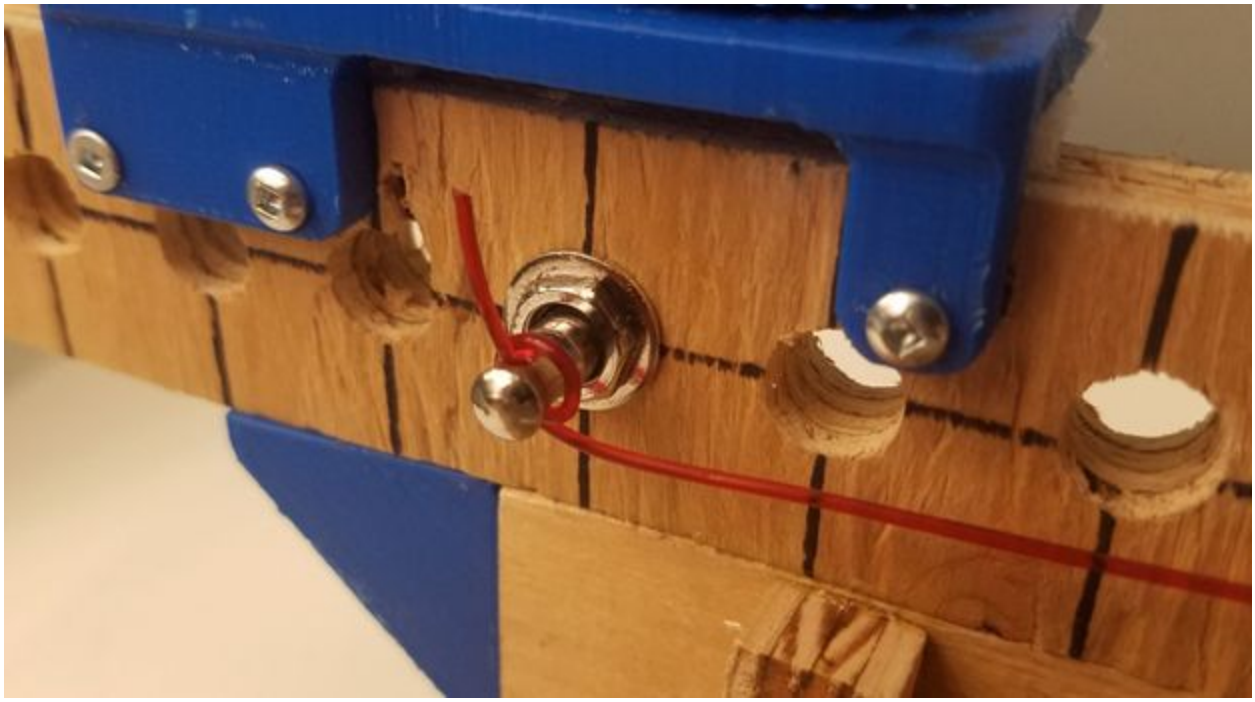
The output from the synchronization of the was the note reproduced by the string, based on the tension at which it was plucked. This note is related to the natural frequency of the string, which is a function of tension. This frequency could be directly measured using a frequency detecting app. We tried many apps but the one which gave us best results was “Note Recognizer” shown in the figure below. Even then, this app would not always give us consistent results. We expect this is due to the quality of our soundboard assembly when compared to professional instrument quality.

C. Parameters

Tuning motor reference position

This parameter was very important to identify early on because we would always go back to it as our “zero position” and the stepper motor would count the number of steps

from this position to every note. The most important aspect of this parameter was that it was easily reproducible and identifiable. Once testing the frequencies for every step, we identified our reference position to be around 147 Hz, based on the Note Recognizer App. We could easily identify this position manually because it was just when the string was starting to slack, there was only one loop of string around the tensioner, the small end of the string was pointing directly upwards, and the gear tick (seen in the figure below) was aligned with the front.



D. Testing and Results

Detailed descriptions of these tests are found in the sections above of this functional requirement, so here is an overview of each test:

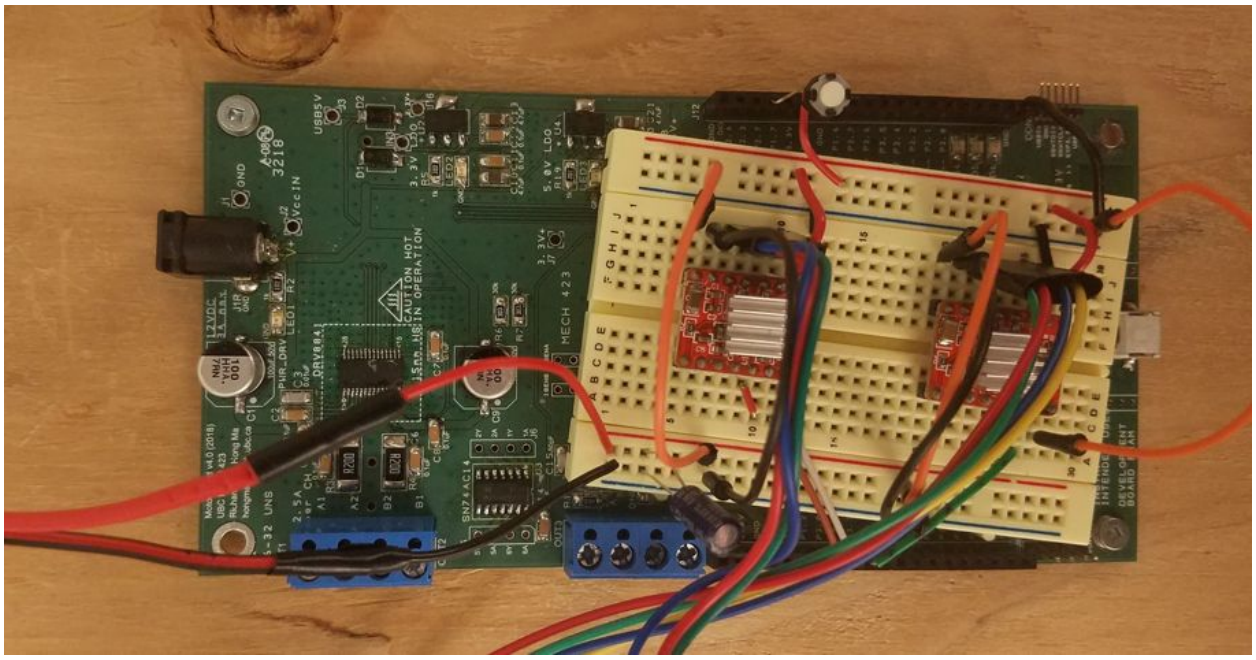
- Tuning motor torque
  - **Objective:**
    Find a motor and motor configuration that can turn the guitar tuning knob enough to reproduce notes an octave above middle C.
  - **Results:**
    After attempting direct connection with the tuning knob with both motors, a 4:1 custom gearbox was created that was able to achieve the notes we desired.

- ## Frequency look-up table
  - **Objective:**
    Make a lookup table that relates motor steps to note frequency, counting the number of gear revolutions and using the Note Recognizer App to identify the frequency.
  - **Results:**

The Note Recognizer App was not as reliable as we expected and gave a mix of frequencies when the string was plucked. This could be due to resonances occurring within our system. However, the App was reliable around 195Hz, which corresponds to the G3 note.

- ## Plucking motor sound
  - **Objective:**
    Configure the motor position and metal pick bend to reproduce quality sound at the notes desired.
  - **Results:**
    Preliminary tests showed that the metal pick was a good approach to produce quality sound. When a trade-off was discovered between high and low tensions, a decision was made to prioritize sound at high tensions to prevent the motor getting stuck in the string.

6. Functional Requirement #3: Plucking + Tuning MSP430 Code

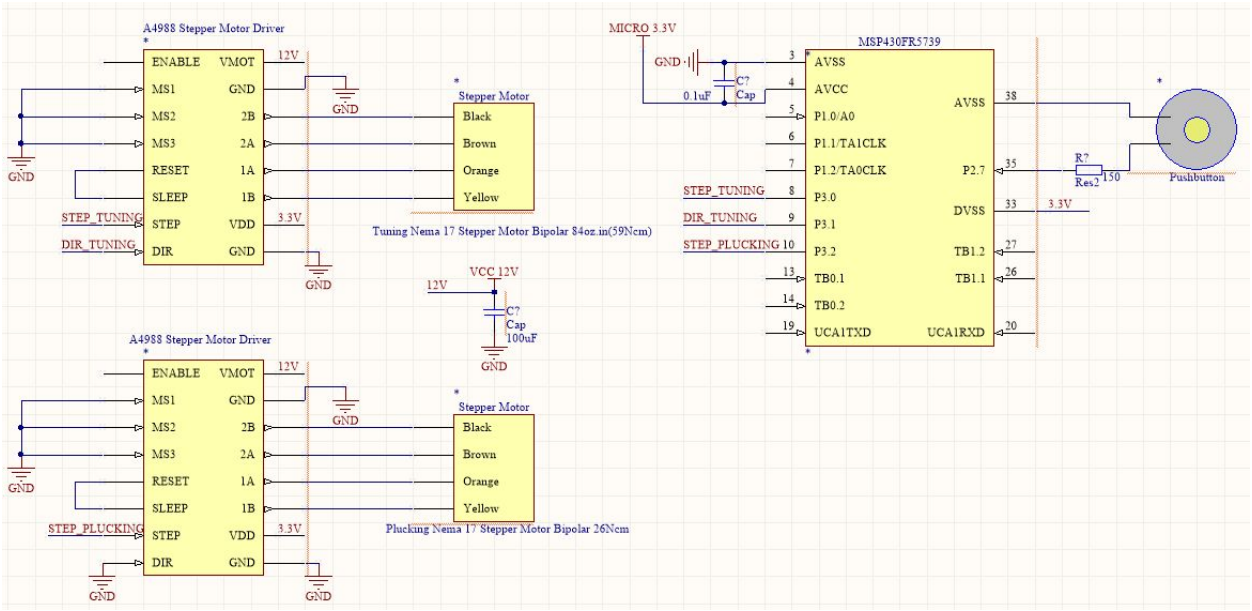


A. Approach and Design

The purpose of the plucking and tuning MSP430 C code is to receive stepper motor absolute position commands from the C# interface and translate them to full step numbers which turn the tuning stepper motor. Once the tuning stepper motor reaches the desired position, it activates the second stepper motor to pluck the string.

The stepper motors are controlled using external Pololu A4988 stepper motor drivers to increase the current limit (upto 5A compared to 2.5A from MSP) for the stepper motors. Both motor drivers are powered using a 12V power supply adapter (the same one used to power the MSP).

The complete electrical schematic diagram is present in the figure below:

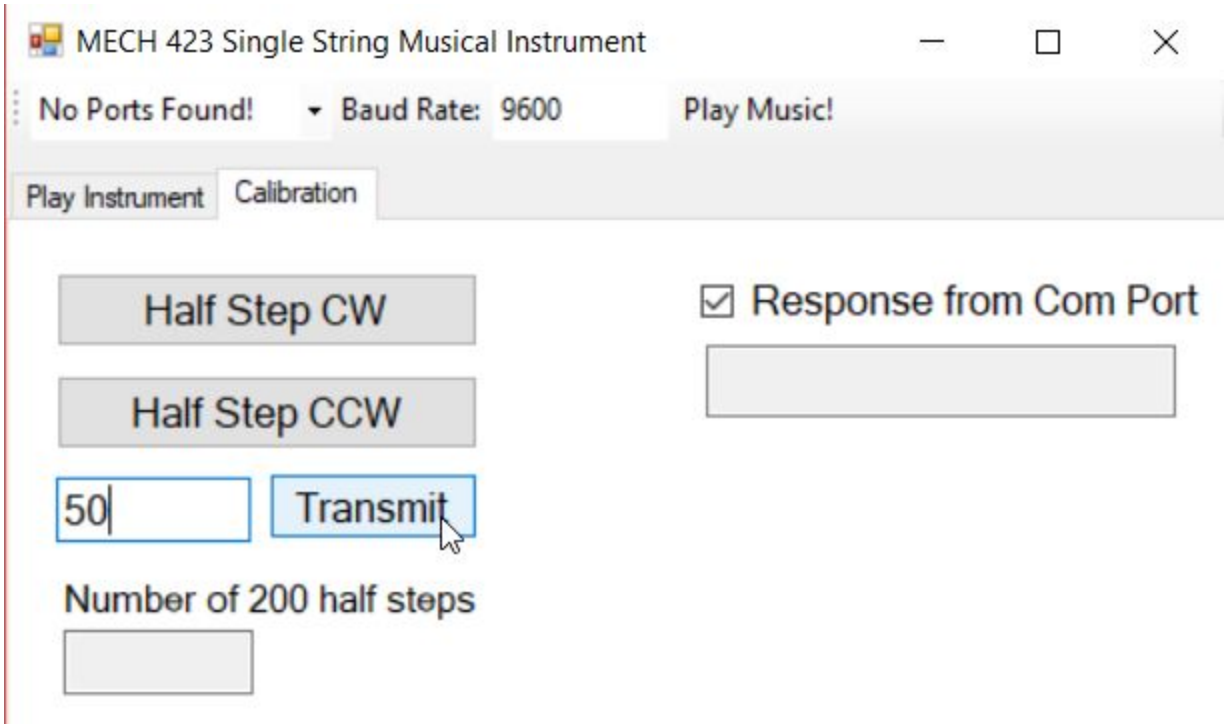


B. Inputs and Outputs

The input into the code is the number of full steps/200 that the tuning motor has to turn by. The output is the revolution of the tuning motor until it reaches the specified step number value followed by the 360 degree revolution of the plucking stepper motor to induce vibration in the string.

The integer value sent from C# to the MSP board is between 0 and 96. This value is then multiplied by 200 in the C code which yields the absolute position in terms of full steps that the tuning stepper motor needs to be at. 1600 full steps correspond to 1 complete revolution of the stepper motor.

When the tuning motor reaches the desired position, it sets the stepperSelect variable value to 1 which activates the plucking motor. Once the plucking motor has turned by the desired number of full steps (400) to complete 1 revolution, it stops spinning and resets the stepperSelect variable to 1 which preps the tuning motor for spinning when the next position command is sent by C# to MSP.



C. Parameters

The parameters of this FR include the following:

1. **Timer B0:** This timer is set in set/reset mode which happens at a frequency determined by setting the value of Timer B0 capture/control register equal to stepperSpeed(or 2) parameter
2. **Stepper Reset(2):** This value determines the position of the tuning stepper motor (or plucking stepper motor) in terms of the number of full steps.
3. **Stepper Count(2):** This variable stores the absolute number of full steps that the tuning motor (or stepper motor) has taken. So if a value less than the current stepper count is input from C#, then the motor loosens the string and if the value is higher than the current stepper count, the the motor tensions the string.
4. **Stepper Speed(2):** It determines the set/reset interrupt frequency of timer B0, thereby determining the speed at which the tuning stepper motor (or plucking stepper motor) turns.
5. **Queue:** Queue of size 1 is created to read the data byte sent from C# to MSP430 and convert it to a stepper count value.

D. Testing and Results

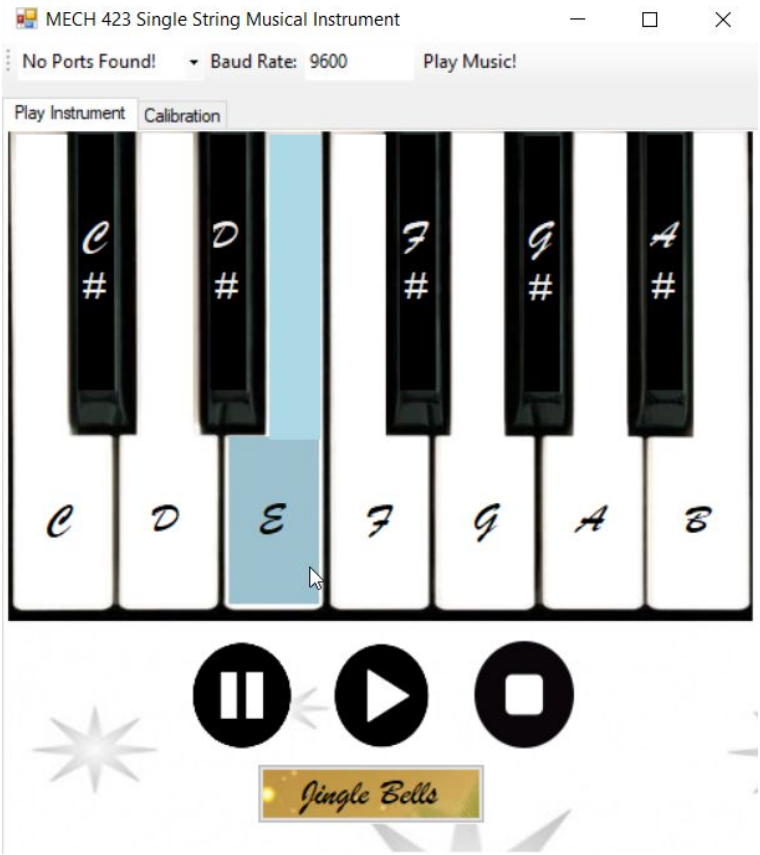
1. **Synchronization of stepper motors:**
   The functioning of the stepper motor was initially tested using a push button connected to P1.7 on MSP430 board. The internal resistor on P1.7 is set to pull-up mode and

pushing the button generates an interrupt. Within the interrupt, a pulse is sent to the STEP input port of the tuning stepper motor driver followed by a pulse to the STEP input port of the plucking stepper motor. This causes the tuning stepper motor to take a half-step and activate the plucking motor.

After successful testing of both stepper motors with the stepper motor drivers, UART communication is enabled between the C# interface and the board which allows the absolute position that the tuning stepper motor needs to be at to be input from C# interface.

2. **Optimization of stepper motor speed:**
   The stepperSpeed and stepperSpeed2 parameters are continuously lowered until the motors are able to turn without stalling. After multiple iterations, the optimized stepperSpeed value is determined to be 700 whereas stepperSpeed2 is 1000. This corresponds to the full stepping frequency of 1428Hz and 1000Hz respectively since the clock is running at 1MHz. At higher frequencies, the tuning motor stalls when it is trying to loosen the string and the plucking motor stalls.

7. Functional Requirement #4: Piano Playing C# Code



A. Approach and Design

Once the user presses different keys on the virtual piano, the stepper motor will tension the string appropriately to play the requested musical note and the plucking motor will engage to strike the harp string. Furthermore, pushing a key on the piano also provides feedback to the user by illuminating that key in blue color.

The design of piano keys consists of the following elements:

1. Picturebox: A piano image showing one octave of notes is inserted inside the picture box.
2. Buttons: Buttons are created on top of the keys in the piano image and a text is placed inside them corresponding to the piano note. The default

background colour of the button is chosen to be either white or black depending on which key it is placed on. A click event is created for each button which sends the pre-set position value to the MSP board, illuminates the key to light blue color, and enables the change keys to default colour timer.

3. Chagne keys to default colour timer: This timer is set with an interval of 1 second and upon the passage of 1 second, the background colours of all the keys are reset to their respective colour.

B. Inputs and Outputs

The input to the system is the press of a piano key by the user. Each key on the piano is linked to a pre-determined half-step position value (determined by tuning the sound and creating a frequency lookup table). The position value is then output to the MSP430 board which turns the tuning motor to the specified position and plucks the string. If another key is pressed before the former key is executed by the instrument, then the latter key overrides the former and the tuning motor plays the note for the latter key.

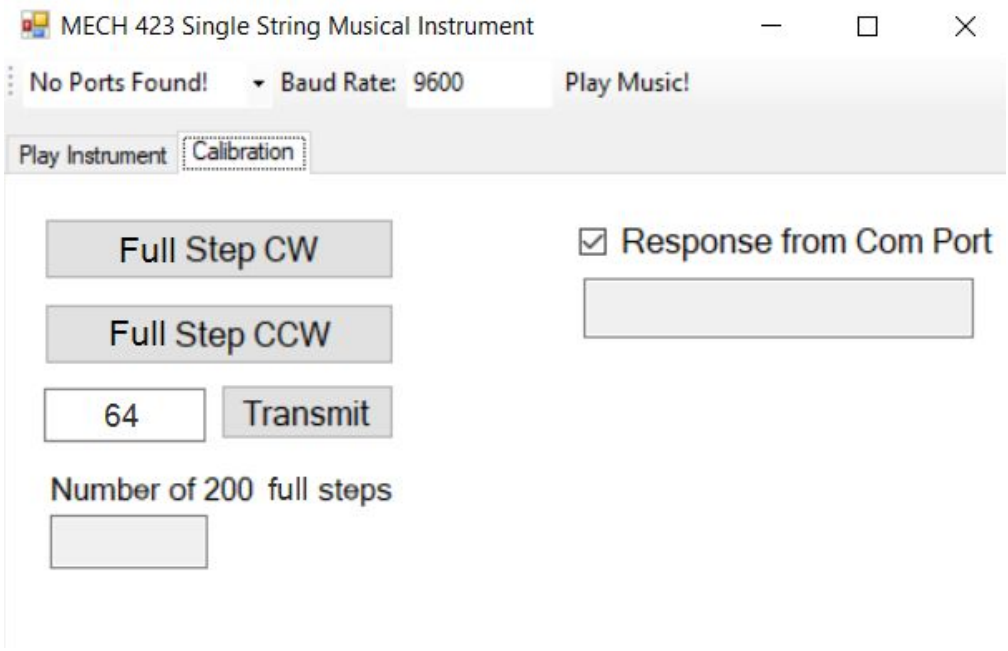
C. Parameters

Following are the parameters of this FR:

1. **Pitch of the played note:** The relative pitch of the played note with respect to the other notes is tuned such that playing notes in succession produces a do-de-ri rhythm.
2. **Illumination time of the pressed key:** The pressed piano key stays illuminated in light blue for 1 second after it has been pushed. If another key is pushed within that 1 second, then it illuminates as well and overrides the command from the previous key.

D. Testing and Results

In order to determine the positions of tuning stepper motor for each note, a calibration tool is developed in C#:



Pushing the *Full Step CW* button turns the tuning motor clockwise by 200 full steps (increases the tension in the string) and *Full Step CCW* turns it counter-clockwise by 200 full steps. The minimum number than can be transmit to the MSP430 board is 0 and the maximum is 96 (which corresponds to 96*200/1600 = 12 revolutions of the tuning motor). Tensioning the string past 12 revolutions could have caused it to break.

Following steps are followed for determining the positions of different musical notes:

1. The power to the stepper motor drivers is disconnected and 0 is transmit to the MSP board from the calibration tab of the C# interface.
2. The tuning motor is manually set to the reference position as described in FR2.
3. The plucker on the plucking motor is set to the vertically downward position.
4. The power is delivered to the stepper motor drivers
5. The motor is stepped to a new position by either clicking on full step button or by transmitting a numerical value using the textbox.
6. After the motor reaches the specified position, the string is plucked and the *Note Recognizer* app is used to detect the musical note generated by the device.

Results:

- Only G3 note perfectly matched with the app repeatedly at step position 64. The reason for other notes not matching with the app might be because there is high tremolo (trembling effect) in the sound produced by the device. Hence, a large variation in frequency over a short period of time.
- Due to time constraints, only octave 3 is tuned by ear by spinning the tuning motor clockwise or counter-clockwise from position 64 (G3 note) and plucking repeatedly.

Following lookup table is produced after tuning octave 3 by ear.

Note	Full step position (scaled down by 200)
C3	48
C3#	52
D3	54
D3#	57
E3	60
F3	64
F3#	66
G3	68
G3#	70
A3	72
A3#	74
B3	76
C4	80

The values in the lookup table above are sent to the MSP board when a piano key is pressed.

8. Functional Requirement #5: Song Playing C# Code

A. Approach and Design

The objective of this code is to provide user the ability to play *Jingle Bells* using on the tunes on the device using a virtual piano on the C# interface. In addition, the user has the ability to play, pause and stop the song while the song sequence is in progress.

The C# interface and backend design for this FR consists of the following:

1. *Jingle Bells* button: Pressing this button initiates the *jingle bells* song sequence and terminates with the resetting of the tuning motor to the reference position. It is a *button* object with a background image embedded inside it.
2. *Pause* button: This button pauses the song while it is being played on the instrument. It is a *picturebox* object with a graphic embedded inside and a click event associated with it.
3. *Play* button: This button resumes playing the song if it was paused. It is a *picturebox* object with a graphic embedded inside and a click event associated with it.
4. *Stop* button: This button stops playing the song if it is in progress and resets the tuning motor to the reference position. It is a *picturebox* object with a graphic embedded inside and a click event associated with it.

The backend design for this FR consists of the following:

- When the *Jingle Bells* button is pressed, a new task is created and started using task.Start() command. This allows other processes to be run concurrently on the C# interface (such as providing the user ability to press pause, play or stop buttons) when the song is being played.

- Within the task, following is the format of commands:
  moveStepperMotor(E3);
  Thread.Sleep(750);
  lock (syncObj) { }

  moveStepperMotor(E3);
  Thread.Sleep(830);
  lock (syncObj) { }
  .
  .
  .
    - moveStepperMotor function moves the tuning motor to the position corresponding to the particular note
    - Thread.Sleep() sleeps the thread for the specified amount of time in milliseconds which gives the tuning motor enough time to traverse to the new position and plucking motor to produce a note.
    - lock(syncObject){} is a *monitor* class object which is entered if the pause button is pressed and exited when the play button is pressed.

B. Inputs and Outputs

The input to the system is the press of the *Jingle Bells, pause, play, and/or stop* button by the user. The output is the initiation of the song sequence, pausing of song, resumption of song (if paused), and termination of the song respectively.

C. Parameters

The parameters for this FR are as follows:

1. **Delay between consecutive notes:** The delay between consecutive notes is minimized by optimizing the speed of both stepper motors as described in the previous FR and by optimizing the sleep times as described in the FR.
2. **Perceptivity of song:** Even though some notes are repeated in *Jingle Bells* (which means that only plucking motor needs to spin), additional delay is introduced even in similar notes to maintain similar time gap between different, thereby increasing user's perceptivity.

D. Testing and Results

The testing for this section was focused on making the instrument reproduce the notes from Jingle Bells and tweaking the delays between each note to match the song as accurately as possible. This was done using a video of a beginner piano tutorial of Jingle Bells on YouTube. The first step was to write down every note from the song, then to use a stopwatch to record the time intervals between each note. These notes and times were translated to the C# program as step UART commands and time delays between each command. Since the delays affected only the C# program, if the delays were not timed right, a command could have been sent to the board before the previous one finished, and it would go to the new note interrupting the previous incomplete one. This brought a long session of tweaking the delay times between each note to give the motor enough time to reach each note.

Results:

We are happy with the final version of Jingle Bells. Originally, the addition of the 4:1 gearbox made the changing between notes so slow we feared that the song would be unrecognizable. But after increasing the motor speed to the maximum possible, and tweaking the delays as optimal as possible, we feel that the song is easily recognized, even with the slight delays between notes.

9. System Evaluation

Each test was described in each specific section in detail. Here is a numbered list of each test in order and their results:

1. **Soundboard assembly sound test**
   Objective:
   Build sound board and test that harp string sound is amplified like theory states.
   Results:
   Once full soundboard was built, it amplified the sound at least twice as loud as without it. We felt happy about it and did not try to improve the soundboard any further.
2. **Motor torque requirement test**
   Objective:
   Find a motor and motor configuration that can turn the guitar tuning knob enough to reproduce notes an octave above middle C.
   Results:
   After attempting direct connection with the tuning knob with both motors, a 4:1 custom gearbox was created that was able to achieve the notes we desired.
3. **Frequency identification**
   Objective:
   Identify the number of steps related to each note frequency using Note Recognizer App, making a lookup table to use later.
   Results:
   Note Recognizer App was unable to give a consistent frequency related to every tension level. It varied between various frequencies. G3 was consistently identified and the frequencies around it as well. This provided a starting point for estimating the step position of every note in octave 3. Although our original scope involved identifying notes over 3 octaves, we had to stick with 1 octave because of heavy time constraints.
4. **Plucking motor test**
   Objective:
   Configure the motor position and metal pick bend to reproduce quality sound at the notes desired.
   Results:
   After multiple attempts, a final configuration was reached which prioritized high frequency notes around the range that we decided to place our octave.
5. **Note tuning**
   Objective:
   Tune the step position of every note more precisely using the half step button on the configuration tool in our C# program.
   Results:

After multiple tweaking iterations of every note, the step positions now resemble their true sound better.

6. **Jingle Bells tuning**
   Objective:
   Tweak delays between every note command to accurately reproduce a beginner version of Jingle Bells.
   Results:
   Delays between note commands were set as the original song time frames between each note. From there every delay was carefully modified and tested to make sure it was as close as possible to the real note time interval. Many times, the time the tuning motor took to change notes made it unable to meet that required time interval. Yet, by the end we were happy that Jingle Bells was easily identified.

10. Reflections

1. What worked and what didn't work? Why? What would you do differently if you could do it again?
Most things worked as planned, except for the tuning motor. Based on a video we found of an autotuner device using our same stepper motor, we expected that these motors would be capable enough to turn the tuning knobs. We were wrong and had to resort to making a custom gearbox for it. If we had more time, we would have used a dc motor with an embedded gearbox and encoder. Position control would have been its own crazy challenge, but it probably would have been able to change the notes at a rate similar to that of the real song.

Furthermore, our goal was to identify notes over multiple octaves instead of just one. However, due to time constraints and higher than expected development time for some of the other design elements of this project, we resorted to fine-tuning one octave. If we were to do things differently, we would ask for the help of a professional musician to help us identify the musical notes more quickly and precisely.

2. Identify 3 things you learned in MECH 423 that you consider the most useful and why.
The three things which we consider to be the most useful are:

1. Firmware design and software integration:
   This course provided an excellent exposure into the field of firmware development and design. After completing this course, we feel confident in developing firmware for any board since we have acquired a very useful skill of reading datasheets of microcontrollers. Furthermore, we learnt how to setup UART communication between laptop and MSP430 microcontroller board - a skill

which can be used for any other microcontroller board as well. We also learnt how to display the incoming data intuitively and process it on a C# interface.

2. Soldering and Electrical circuit design for signal amplification, rectification and high current delivery:
   Soldering small components is a very useful skill which we gained because it would allow us to quickly create prototype electronic boards for testing and replace broken electronic components without relying on the help from technicians.

   In the lectures by Dr. Hong following the completion of lab 3, we learnt about various circuit design strategies involving op-amps, photodiodes, RTDs etc to improve the output signal quality as well as amplify the signals for data processing. Furthermore, we learnt about how low- and high-side MOSFETs deliver power to the load and how parasitic capacitance can affect the performance of the MOSFETs. We also learnt about the pros and cons of BJTs and which applications they are best suited for.

3. Mechatronic Product Development Timeline and Risk Estimation:
   During the completion of the final project, some design elements (for example, soundboard design, tuning motor attachment and piano C# interface design) took longer than expected which did not leave us with enough time to identify notes over multiple octaves. After completing the final project, we have realized that time overruns can be avoided if a generous amount of buffer time is added to each design element of a mechatronic product.

3. What are some limits of your knowledge and expertise as a mechatronic engineer? Identify 3 things you would like to learn going forward. What is your strategy to acquire knowledge in these areas?

In this course, the firmware involved setting individual bits. However, for microcontrollers such as STM32, there are libraries with pre-defined functions which allow setting up ports and interrupts with a shorter amount but more blackbox type of code. The latter is what is used in the industry though as it is more readable. Moreover, CAN, SPI, and I2C are very important communication protocols which were not covered in this course but are widely used in the industry. The best way to acquire these missing firmware skills is to join a student design team such as Formula Electric.

In terms of electrical circuit design, we know how to rectify the output signals using various electronic components. However, we do not know how to design schematics and PCBs in industry leading software such as Altium and Eagle. The PCB design skill can

be acquired by watching video tutorials on youtube and/or by joining a UBC student design team which designs its own PCBs to accomplish tasks.
Finally, thermal cooling of and heat sinking from the electronic components present on high-current PCBs is also very important as temperatures affect the performance of the electronic circuit. Even though there was one third year course (MECH375) which we took on heat theory, we did not get a chance to apply that knowledge practically. This skill gap can be filled by working in a power electronics department of a company and/or by joining a student design team.

11. References

Waltham, C., Kotlicki, A., Dunwoody, L., Lee, T., Lin, J., & Lin, B. (2008). Vibrational Characteristics of Harp Soundboards. *The Journal of the Acoustical Society of America, 123*(5), 3381-3381. doi:10.1121/1.2934022

APPENDIX A: FR3 PLUCKING + TUNING MOTOR CODE

```
#include <msp430.h>
#include <msp430fr5739.h>
#include <stdio.h>

/**
 * main.c
 */



//MECH423 FINAL PROJECT MASTER TUNING MOTOR CODE
#define FALSE 0
#define TRUE 0

unsigned int stepperState = 0; //1-4 states
unsigned int stepperSpeed = 700;
unsigned int stepperSpeed2 = 1000;

unsigned int stepperCount = 0;
unsigned int stepperReset = 0; //How many steps before reset
//unsigned int stepperReset = 1600; //How many steps before reset
unsigned int stepperEnable = 1;

unsigned int stepperSelect = 0; //0-tuning motor, 1-plucking motor

unsigned int stepperCount2 = 0;
unsigned int stepperReset2 = 400; //How many steps before reset

unsigned long int noteNumber = 0;


//Code for initializing the queue and defining queue functions
typedef struct{
    unsigned int front;
    unsigned int num;
    unsigned int capacity;
    unsigned int* arr;
} Queue;

void initQueue(Queue* q, unsigned int numValues) {
    q->front = 0;
    q->num= 0;
    q->capacity = numValues; // or some other value
    q->arr= (unsigned int*) malloc(q->capacity * sizeof(unsigned int));
}

unsigned int isEmpty(Queue* q) {
    if(q->num== 0)
        return TRUE;
    else
```

```
        return FALSE;
}

unsigned int isFull(Queue* q) {
    if(q->num== q->capacity)
        return TRUE;
    else
        return FALSE;
}

unsigned int enqueue(Queue* q, int val) {
    if(isFull(q))
        return FALSE;
    else{
        q->arr[(q->front + q->num) % q->capacity] = val;
        q->num++;
        return TRUE;
    }
}

unsigned int dequeue(Queue* q) {
    int val;
    if(isEmpty(q))
        return FALSE;
    else{
        val = q->arr[q->front];
        q->arr[q->front] = -1;
        q->front = (q->front + 1) % q->capacity;
        q->num--;
        return  val;
    }
}

unsigned int sizeQueue(Queue *q){
    return q->num;
}

void inline printQueue(Queue *q){
    unsigned int index = 0;
    int arrVal = 0;

    while(q->arr[index] != NULL){
        arrVal = q->arr[index];
        if (arrVal < 0)
            index++;
        else{
            while (!(UCA1IFG & UCTXIFG));
            UCA1TXBUF = arrVal;
            index++;
        }
    }
}

void printFullQueueError(void){
    /*int j;

    for (j = 0; fullQueueErrorMsg[j] != 0; j++){
        while (!(UCA1IFG & UCTXIFG));
        UCA1TXBUF = fullQueueErrorMsg[j];
    }*/
```

```
}

//TUNING STEPPER MOTOR FUNCTIONS
void moveStepperToPosition(int positionNum){
   //while(stepperCount != stepperReset){
      TB0CCTL1 = OUTMOD_3 + CCIE;

}

void sendPulseToSlave(void){
   //This timer B2 sends a pulse to P2.7 of the slave board every 1 second to initiate the plucking motor
   //Set up P3.6 to output a pulse
   /*
   P3DIR |= BIT6;
   P3SEL1 &= ~BIT6;
   P3SEL0 |= BIT6;
   TB2CCR0 = 6000-1;                       // PWM Period
   TB2CCTL1 = OUTMOD_7;                    // CCR1 reset/set
   TB2CCR1 = 500;                          // CCR1 PWM duty cycle
   TB2CTL |= TBSSEL_2 + MC_1 + TBCLR + ID__8;        // SMCLK, up mode, clear TAR (clears the timer B
count)
   P3DIR &= ~BIT6;
   */
   P3OUT ^= BIT6;
}

void moveTuningMotor(int noteNum){
   /* C1 = 1
    * C1# = 2
    * D1 = 3
    * D1# = 4
    * E1 = 5
    * F1 = 6
    * F1# =7
    * G1 =8
    * G1# =9
    * A1 =10
    * A1# =11
    * B1 =12
    * C2 =13
    * C2# =14
    * D2 =15
    * D2# =16
    * E2 =17
    * F2 =18
    * F2# =19
    * G2 =20
    * G2# =21
    * A2 =22
    * A2# =23
    * B2 =24
    * C3 =25
    * C3# =26
    * D3 =27
    * D3# =28
    * E3 =29
    * F3 =30
    * F3# =31
    * G3 =32
    * G3# =33
```

```
    * A3 =34
    * A3# =35
    * B3 =36
    * C4 =37
    * C4# =38
    * D4 = 39
    * D4# =40
    * E4 =41
    * F4 =42
    * F4# =43
    * G4 =44
    * G4# =45
    * A4 =46
    * A4# =47
    * B4 =48
    * C5 =49
    * C5# =50
    * D5 =51
    * D5# =52
    * E5 =53
    * F5 =54
    * F5# =55
    * G5 =56
    * G5# =57
    * A5 =58
    * A5# =59
    * B5 =60
    */

   //playnote routine
   stepperReset = noteNum;
   moveStepperToPosition(noteNumber);
}


Queue myq;

int main(void)
{
   WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer

   initQueue(&myq, 1); // initialize the queue with size 1

   CSCTL0 = 0xA500;                        // Write password to modify CS registers
   CSCTL1 = DCOFSEL0 + DCOFSEL1;           // DCO = 8 MHz
   CSCTL2 = SELM0 + SELM1 + SELA0 + SELA1 + SELS0 + SELS1;              //remove divider to take SMCLK to
8Mhz

   //CONFIGURE UART PORTS
   P2SEL0 &= ~(BIT5 + BIT6);
   P2SEL1 |= BIT5 + BIT6;
   //Configure UART 1
   UCA1CTLW0 = UCSSEL0;                                    // Run the UART using ACLK
   //UCA0MCTLW = UCOS16 + UCBRF_1 + 0x4900;                  // Baud Rate = 9600; UCOS16 = 1; UCBR0 = 52;
UCBRF0 = 1; UCBRS0 = 0x4900
   UCA1MCTLW = UCOS16 + UCBRF_10 + 0xF700;                  // Baud Rate = 57600; UCOS16 = 1; UCBR0 =
52; UCBRF0 = 10; UCBRS0 = 0xF700
   UCA1BRW = 52;
   UCA1IE |= UCRXIE;                                   // Enable UART Rx interrupt
```

```
    P2DIR &= ~BIT7;                  //Make 1.7 input
    P2REN |= BIT7;                   //Enable 1.7s Resistors
    P2OUT |= BIT7;                   //Enable 1.7 Resistor to pullup
    P2IES |= BIT7;                  //Interrupt 1.7 from falling edge
    P2IE |= BIT7;                   //Enable interrupts


    //A4988 Driver
    P3DIR |= 0b01001111;
    P3SEL1 = 0;
    P3SEL0 = 0;
    P1DIR |= 0b00110000;
    P1SEL0 |= 0b00000000;
    P1SEL1 |= 0;
    P1OUT |= 0b00100000;


    //Setup Interrupt which manages the velocity of the stepper

    TB0CTL = TBSSEL_2 + ID__1 + MC_1;   //Set clock to read from SMCLK, divide by 1, operate in up mode
    TB0CCR0 = stepperSpeed;
    TB0CCR1 = stepperSpeed *0.5;
    TB0CCTL1 = OUTMOD_3;
    TB0CCTL1 &= ~CCIE;
    //TB0CCTL1 = OUTMOD_3;


    __enable_interrupt();            //enable Global interrupts

    while(1){
        //while (!(UCA1IFG & UCTXIFG));
        //UCA1TXBUF = 10;

        if(sizeQueue(&myq) >= 1 && sizeQueue(&myq) <= 3){
            noteNumber = dequeue(&myq);
            noteNumber = noteNumber* 200;
            moveTuningMotor(noteNumber);
        }
        _delay_cycles(100000);
    }

    return 0;
}

#pragma vector = PORT2_VECTOR;
__interrupt void PORT2_ISR(void)
{
    P2IFG &= ~BIT0;             //reset interrupt
    stepperEnable = 1;
    TB0CCTL1 = OUTMOD_3 + CCIE;
   // moveTuningMotor(noteNumber);
    P2OUT ^= 0b00000001;
}


#pragma vector = TIMER0_B1_VECTOR
__interrupt void TIMER0_B1_ISR(void){

    if(stepperCount < stepperReset){
```

```
        P3OUT &= ~BIT1;
        stepperCount++;
        P3OUT ^= BIT0;
    }

    else if(stepperCount > stepperReset){
        P3OUT |= BIT1;
        stepperCount--;
        P3OUT ^= BIT0;
    }

    if (stepperCount == stepperReset){
        //stepperCount = 0;

        //TB0CCTL1 = OUTMOD_3;
        stepperSelect = 1;
    }

    //now plucking motor starts
    if (stepperSelect == 1){
            TB0CCR0 = stepperSpeed2;
            TB0CCR1 = stepperSpeed2*0.5;
            stepperCount2++;

            if (stepperCount2 >= stepperReset2){
                stepperCount2 = 0;
                stepperSelect = 0;
                stepperEnable = 0;
                TB0CCTL1 = OUTMOD_3;
                TB0CCR0 = stepperSpeed;
                TB0CCR1 = stepperSpeed*0.5;
            }
            else{
                P3OUT ^= BIT2;
            }
    }

    TB0CCTL1 &= ~(0x1); //clear the timer B0 flag
}

// Receive data from C#
#pragma vector = USCI_A1_VECTOR
__interrupt void USCI_A1_ISR(void)
{
    // Get the new byte from the Rx buffer
    unsigned int RxByte = 0;                // can also be int type
    RxByte = UCA1RXBUF;                     // Get the new byte from the Rx buffer

    if (!isFull(&myq)){
       enqueue(&myq, RxByte);
    }
    else{
       printFullQueueError();
    }

    if(isFull(&myq)){
        printQueue(&myq);
    }
}
```

APPENDIX B: FR4 and FR5 C# CODE

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.IO.Ports;
using System.Collections.Concurrent;
using System.Media;
using System.Threading;
using System.Threading.Tasks;

namespace MECH423_Lab2_Exam_Serial_Comm
{


    public partial class Form1 : Form
    {
        int currentStep = 0;
        int NUM_STEPS_PER_COMMAND = 1;
        int numFullStepsPerHalfStepComand = 200;
        int NUM_STEPS_PER_REV = 1600;
        int MAX_REVS = 12;

        int maxNumCommands;

        string C3 = "48";
        string C3sharp = "50";
        string D3 = "52";
        string D3sharp = "54";
        string E3 = "56";
        string F3 = "60";
        string F3sharp = "62";
        string G3 = "64";
        string G3sharp = "66";
        string A3 = "68";
        string A3sharp = "72";
        string B3 = "76";
        string C4 = "80";

        int veryShortInterval = 750;
        int shortInterval = 1500;
        int mediumInterval = 3000;
        int longInterval = 6000;

        int traverseTime4Steps = 625;

        bool pauseFlag = false;
        bool playFlag = true;
        bool stopFlag = false;
```

```
private object syncObj = new object();

//Bitmap bmp, bmpPiano;
//Graphics g, gAlarm;

List<double> velocityList = new List<double>();
byte[] TxBytes = new Byte[3];

public Form1()
{
   maxNumCommands = NUM_STEPS_PER_REV * MAX_REVS / numFullStepsPerHalfStepComand;
   InitializeComponent();
}

private void ComPortUpdate()
{
   cmbComPort.Items.Clear();
   string[] comPortArray = System.IO.Ports.SerialPort.GetPortNames().ToArray();
   Array.Reverse(comPortArray);
   cmbComPort.Items.AddRange(comPortArray);
   if (cmbComPort.Items.Count != 0)
      cmbComPort.SelectedIndex = 0;
   else
      cmbComPort.Text = "No Ports Found!";
}

private void Form1_Load(object sender, EventArgs e)
{
    /*

   bmpPiano = new Bitmap(947, 310);
   //pbPiano.Controls.Add(pbAlarmHands);
   pbPiano.Location = new Point(0, 0);
   pbPiano.BackColor = Color.Transparent;
   gAlarm = Graphics.FromImage(bmpPiano);
   */
   lblIncomingDataRate.Visible = false;
   chkShowResponse.Checked = true;
   ComPortUpdate();
}

private void btnConnect_Click(object sender, EventArgs e)
{
   if (btnConnect.Text == "Play Music!")
   {
      if (cmbComPort.Items.Count > 0)
      {
         try
         {
            serialPort1.BaudRate = Convert.ToInt16(txtBaudRate.Text);
            serialPort1.PortName = cmbComPort.SelectedItem.ToString();
            serialPort1.Open();
            btnConnect.Text = "Disconnect";
            timer1.Enabled = true;
            changeKeysToDefaultColor.Enabled = true;
            lblIncomingDataRate.Visible = true;
         }
         catch (Exception ex)
         {
            MessageBox.Show(ex.Message);
```

```
            }
        }
    }
    else
    {
        try
        {
            serialPort1.Close();
            btnConnect.Text = "Play Music!";
            timer1.Enabled = false;
            changeKeysToDefaultColor.Enabled = false;
            lblIncomingDataRate.Visible = false;
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
}

private void genericTextBoxEventHandler(object sender, EventArgs e)
{
    TextBox currentTextBox = sender as TextBox;
    short parseResult;
    if (Int16.TryParse((currentTextBox.Text), out parseResult))
    {
        if (parseResult > maxNumCommands)
            parseResult = (short)maxNumCommands;
        if (parseResult <= 0)
            parseResult = 0;
        currentTextBox.Text = parseResult.ToString();
    }
    else
        currentTextBox.Text = "";
}

void moveStepperMotor(string numHalfSteps)
{
    currentStep = Convert.ToInt32(numHalfSteps);
    TxBytes[0] = Convert.ToByte(numHalfSteps);

    if(currentStep <= maxNumCommands)
    {
        try
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Write(TxBytes, 0, NUM_STEPS_PER_COMMAND);
                txtNumHalfSteps.Text = numHalfSteps;
            }
        }
        catch (Exception Ex)
        {
            MessageBox.Show(Ex.Message);
        }
    }

}

/// <summary>
```

```
/// Creates a plot of the accelerometer data
/// </summary>
/// <param name="position"></param>
/// <param name="velocity"></param>
private void updatePlot(double positionChart, double velocityChart)
{

}


private void serialPort1_DataReceived(object sender, System.IO.Ports.SerialDataReceivedEventArgs e)
{
   while (serialPort1.IsOpen && serialPort1.BytesToRead != 0)
   {

      int currentByte = serialPort1.ReadByte();

      if (chkShowResponse.Checked)
         this.BeginInvoke(new EventHandler(delegate
         {
            txtRawSerial.AppendText(currentByte.ToString() + ", ");
         }));
   }
}



private void btD3_Click(object sender, EventArgs e)
{
   moveStepperMotor("22");
}

private void btE3_Click(object sender, EventArgs e)
{
   moveStepperMotor("33");
}

private void btHalfStepCW_Click(object sender, EventArgs e)
{
   currentStep += NUM_STEPS_PER_COMMAND;
   moveStepperMotor(Convert.ToString(currentStep));
}

private void btHalfStepCCW_Click(object sender, EventArgs e)
{
   if (currentStep > 0)
      currentStep  -= NUM_STEPS_PER_COMMAND;

   moveStepperMotor(Convert.ToString(currentStep));
}

private void btTransmitHalfSteps_Click(object sender, EventArgs e)
{
   moveStepperMotor(Convert.ToString(tbHalfSteps.Text));
}

private int playJingleBells()
{
   //while(processing)
   //Song Start
```

```
moveStepperMotor(E3);
Thread.Sleep(10000);
lock(syncObj) { }

moveStepperMotor(E3);
Thread.Sleep(750);
lock (syncObj) { }

moveStepperMotor(E3);
Thread.Sleep(830);
lock (syncObj) { }

moveStepperMotor(E3);
Thread.Sleep(1760);
lock (syncObj) { }

moveStepperMotor(E3);
Thread.Sleep(830);
lock (syncObj) { }

moveStepperMotor(E3);
Thread.Sleep(830);
lock (syncObj) { }

moveStepperMotor(E3);
Thread.Sleep(1760);
lock (syncObj) { }

moveStepperMotor(E3);
Thread.Sleep(550);
lock (syncObj) { }

moveStepperMotor(G3);
Thread.Sleep(1750);
lock (syncObj) { }

moveStepperMotor(C3);
Thread.Sleep(3250);
lock (syncObj) { }

moveStepperMotor(D3);
Thread.Sleep(1500);
lock (syncObj) { }

moveStepperMotor(E3);
Thread.Sleep(2550);
lock (syncObj) { }

moveStepperMotor(F3);
Thread.Sleep(1650);
lock (syncObj) { }

moveStepperMotor(F3);
Thread.Sleep(910);
lock (syncObj) { }

moveStepperMotor(F3);
Thread.Sleep(940);
```

```
        lock (syncObj) { }

        moveStepperMotor(F3);
        Thread.Sleep(940);
        lock (syncObj) { }

        moveStepperMotor(F3);
        Thread.Sleep(540);
        lock (syncObj) { }

        moveStepperMotor(E3);
        Thread.Sleep(1650);
        lock (syncObj) { }

        moveStepperMotor(E3);
        Thread.Sleep(760);
        lock (syncObj) { }

        moveStepperMotor(E3);
        Thread.Sleep(510);
        lock (syncObj) { }

        moveStepperMotor(E3);
        Thread.Sleep(510);
        lock (syncObj) { }

        moveStepperMotor(E3);
        Thread.Sleep(550);
        lock (syncObj) { }

        moveStepperMotor(D3);
        Thread.Sleep(1500);
        lock (syncObj) { }

        moveStepperMotor(D3);
        Thread.Sleep(570);
        lock (syncObj) { }

        moveStepperMotor(E3);
        Thread.Sleep(1000);
        lock (syncObj) { }

        moveStepperMotor(D3);
        Thread.Sleep(1500);
        lock (syncObj) { }

        moveStepperMotor(G3);
        Thread.Sleep(10000);
        lock (syncObj) { }

        moveStepperMotor("0");

        return 0;
    }

    private void btJingleBells_Click(object sender, EventArgs e)
    {
        Task<int> task = new Task<int>(playJingleBells);
        task.Start();
    }
```

```
private void timer1_Tick(object sender, EventArgs e)
{

}

private void changeKeysToDefaultColor_Tick(object sender, EventArgs e)
{
    btC3.BackColor = Color.White;
    btC3_top.BackColor = Color.White;
    btD3.BackColor = Color.White;
    btD3_top.BackColor = Color.White;
    btE3.BackColor = Color.White;
    btE3_top.BackColor = Color.White;
    btF3.BackColor = Color.White;
    btF3_top.BackColor = Color.White;
    btG3.BackColor = Color.White;
    btG3_top.BackColor = Color.White;
    btA3.BackColor = Color.White;
    btA3_top.BackColor = Color.White;
    btB3.BackColor = Color.White;
    btB3_top.BackColor = Color.White;

    btCsharp.BackColor = Color.Black;
    btDsharp.BackColor = Color.Black;
    btFsharp.BackColor = Color.Black;
    btGsharp.BackColor = Color.Black;
    btAsharp.BackColor = Color.Black;
    changeKeysToDefaultColor.Enabled = false;
}

private void btC3_Click(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btC3.BackColor = Color.LightBlue;
    btC3_top.BackColor = Color.LightBlue;
    moveStepperMotor(C3);
}
private void btD3_Click_1(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btD3.BackColor = Color.LightBlue;
    btD3_top.BackColor = Color.LightBlue;
    moveStepperMotor(D3);
}

private void btE3_Click_1(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btE3.BackColor = Color.LightBlue;
    btE3_top.BackColor = Color.LightBlue;
    moveStepperMotor(E3);
}

private void btF3_Click(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btF3.BackColor = Color.LightBlue;
    btF3_top.BackColor = Color.LightBlue;
    moveStepperMotor(F3);
```

```
}

private void btG3_Click(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btG3.BackColor = Color.LightBlue;
    btG3_top.BackColor = Color.LightBlue;
    moveStepperMotor(G3);
}

private void btA3_Click(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btA3.BackColor = Color.LightBlue;
    btA3_top.BackColor = Color.LightBlue;
    moveStepperMotor(A3);
}

private void btB3_Click(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btB3.BackColor = Color.LightBlue;
    btB3_top.BackColor = Color.LightBlue;
    moveStepperMotor(B3);
}

private void btC4_Click(object sender, EventArgs e)
{
    moveStepperMotor(C4);
}

private void btCsharp_Click(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btCsharp.BackColor = Color.LightBlue;
    moveStepperMotor(C3sharp);
}

private void btDsharp_Click(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btDsharp.BackColor = Color.LightBlue;
    moveStepperMotor(D3sharp);
}

private void btFsharp_Click(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btFsharp.BackColor = Color.LightBlue;
    moveStepperMotor(F3sharp);
}

private void btGsharp_Click(object sender, EventArgs e)
{
    changeKeysToDefaultColor.Enabled = true;
    btGsharp.BackColor = Color.LightBlue;
    moveStepperMotor(G3sharp);
}

private void btAsharp_Click(object sender, EventArgs e)
```

```
        {
            changeKeysToDefaultColor.Enabled = true;
            btAsharp.BackColor = Color.LightBlue;
            moveStepperMotor(A3sharp);
        }

        private void pbPause_Click(object sender, EventArgs e)
        {
            if (pauseFlag == false)
            {
                // This will acquire the lock on the syncObj,
                // which, in turn will "freeze" the loop
                // as soon as you hit a lock(syncObj) statement
                Monitor.Enter(syncObj);
                pauseFlag = true;
            }
        }

        private void pbPlay_Click(object sender, EventArgs e)
        {
            if (pauseFlag)
            {
                pauseFlag = false;
                // This will allow the lock to be taken, which will let the loop continue
                Monitor.Exit(syncObj);
                playFlag = true;
            }
        }

        private void pbStop_Click(object sender, EventArgs e)
        {
            stopFlag = true;
        }
    }
}
```