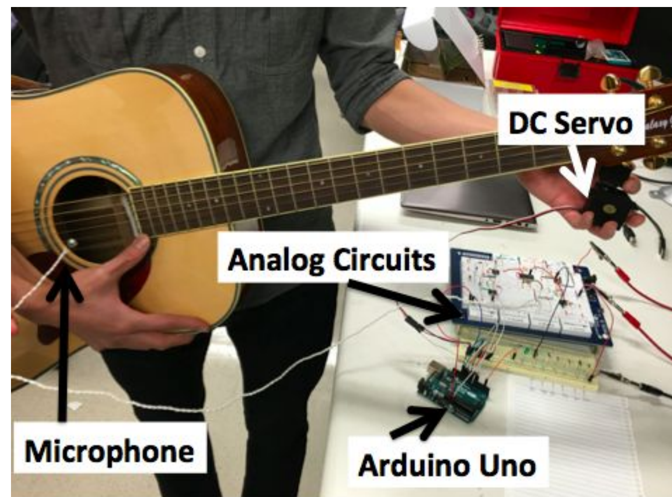
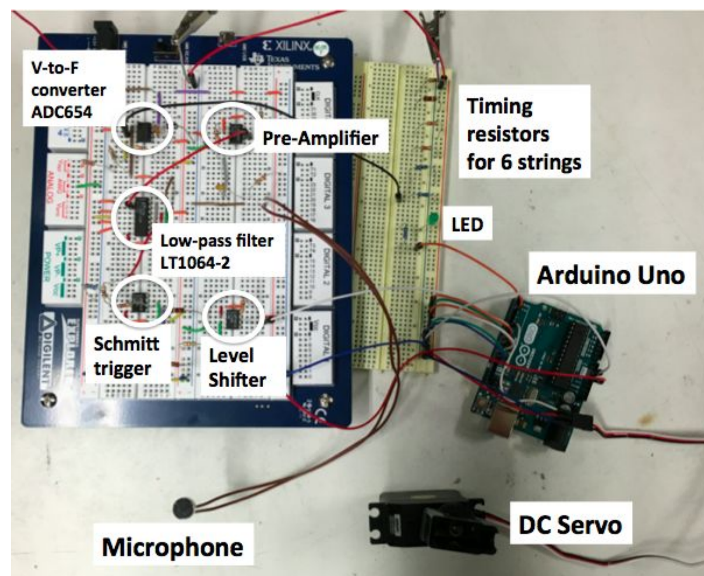
Automatic Guitar Tuner

Justin Doong, Minmin Hou, Wendy Li

Project Overview:
In this project, we designed and built an automatic guitar tuner. When using the tuner, the user plucks and tunes the strings one by one. The user tells the tuner which string to tune using mechanical switches. A green LED will be lit to indicate the selected string is tuned. The prototype is shown below in Figure 1.



(a) Guitar tuner system overview



(b) Close-up look at circuit components
Figure 1. Automatic Guitar Tuner prototype

Component List:
1 electret microphone
1 voltage-to-frequency converter AD654
1 switched-capacitor low-pass filter LTC1064-2
3 operational amplifiers LTC1056
Multiple resistors (values in subsequent sections)
Multiple capacitors (values in subsequent sections)
1 Arduino Uno
1 DC servo

System Design:
The block diagram of the entire system is shown below in Figure 2. The complete system can be divided into four sub-systems: microphone input, filter, analog frequency counter, microcontroller, and DC servo with guitar peg attachment.

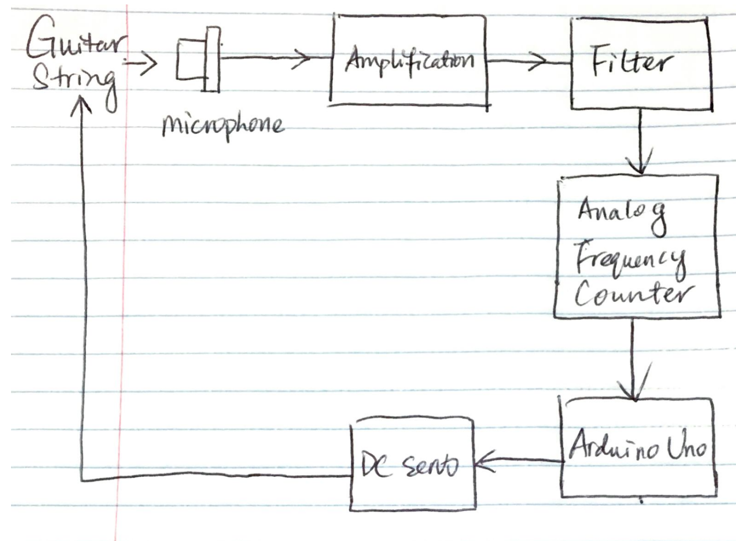


Figure 2. Block diagram of Automatic Guitar Tuner

Microphone input
We used an electret microphone to transduce guitar audio signals into electrical signals. The output of the microphone is connected to a 1uF capacitor. The pull-up resistor and the capacitor filter out frequency components lower than the lowest frequency of the guitar strings (here we chose 30Hz as the lowest string frequency is about 80Hz). The signal is then further amplified by an inverting amplification circuit built using an LTC1056 op-amp. The schematic is shown below in Figure 3.

The signal needs amplification because the signal directly from the microphone is not large enough for the subsequent signal processing circuits. The values of R1 and R2 are 1k and 4.7M ohm, respectively.

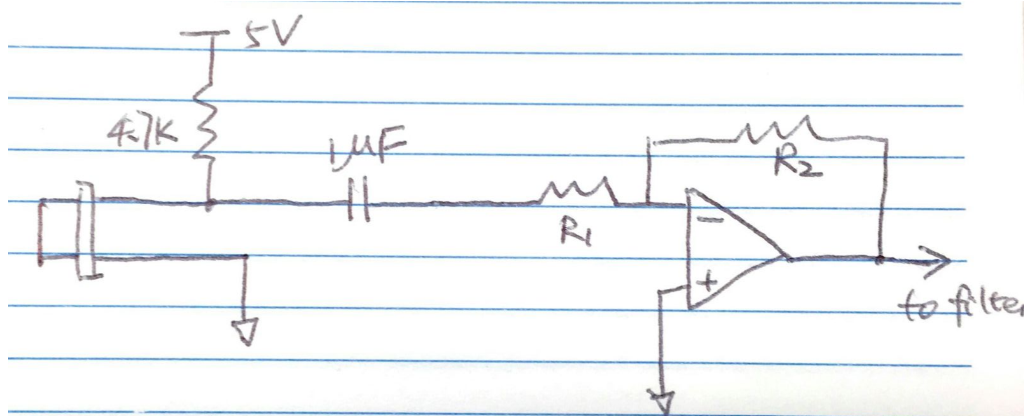


Figure 3. Schematic of microphone input circuit

Filter

The amplified guitar signal is then filtered with a switched-capacitor (SC) low-pass filter (LPF). We used an 8th-order Butterworth SC LPF IC chip, the LTC1064-2. The cutoff frequency of the switched-capacitor filter depends on which string is being tuned. The filter cutoff frequency is determined by the input frequency to the LTC1064 chip, which is generated by a voltage-to-frequency converter, the AD654. The circuit schematic is shown below in Figure 4. To be able to vary the output frequency of the AD654 (and thus also the LTC1064 cutoff frequency), we did the following:

1) Fixed input voltage and timing capacitor values:
   We set the input voltage Vin to be 0.5V, which is obtained through a resistor divider from the 5V power supply. We chose 1nF for the timing capacitor (CT). We chose values that would allow us to use readily-available resistor values, and that would limit the charging current going into the timing capacitor to be within the range specified on the datasheet of the AD654.
2) Calculated the value of the timing resistor (RT) for each guitar string:
   The values can be found in the table below. We calculated the resistor values based on fclk:fc = 100:1 for the switched-capacitor chip (set by tying Pin 10 of LTC1064 -5V).

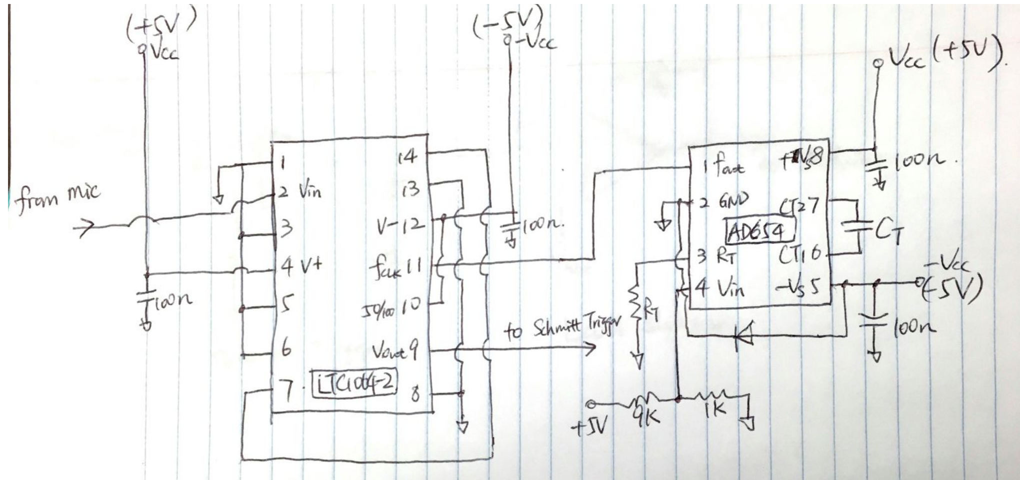


Figure 4. Schematic of low pass filter built with LTC1064-2 and AD654

Table 1. Timing resistor values for the six guitar strings

Guitar string	E	A	D	G	B	e
Frequency (Hz)	82.41	110	146.83	196	246.94	329.63
Rt (ohm) max	6.07k	4.55k	3.41k	2.55k	2.02k	1.52k
Rt (ohm) min	4.55k	3.41k	2.55k	2.02k	1.52k	
Rt use(ohm)	5.1k	3.9k	3k	2.2k	1.8k	1.3k

The output from the switched-capacitor filter chip is a sine wave because the second and higher harmonics of the guitar signals are attenuated significantly by the 8th-order Butterworth filter.

Analog frequency counter

The analog frequency counter converts the sine wave output from the filter to a square wave and shifts the DC offset of the wave into the range readable by the Arduino.

The schematic of the analog frequency counter is shown in Figure 5 below. It consists of a Schmitt trigger and a DC level shifter. The two triggering voltages are (R1 / R2) * Vz, which are determined by the Zener diode voltage (Vz) and the resistor values. We want the triggering voltages to be about +/- 0.5V because the output from the filter is about +/- 0.8V for the lowest-frequency string (and becoming larger when string frequency increases). We used 3.3V Zener diodes. To get +/- 0.5V triggering voltages, we chose R1 = 1k and R2 = 6.1k.

The level shifter is based on an op-amp. We want the input to the Arduino to be within the range 0 - 5 V, but the output of the Schmitt trigger is -5V to 5V.
For an inverting op-amp, (Vin - VREF) / R3 = - (Vout - VREF) / R4
And given the conditions that Vin = -5V → Vout = 5V and Vin = 5V → Vout = 0V
We can get that VREF = 1.4V and R4 / R3 = 0.75. We chose R3= 3.9k, R4 = 3k.
VREF is obtained by a voltage divider from 5V, the two resistors are 5.1k and 1.5k.

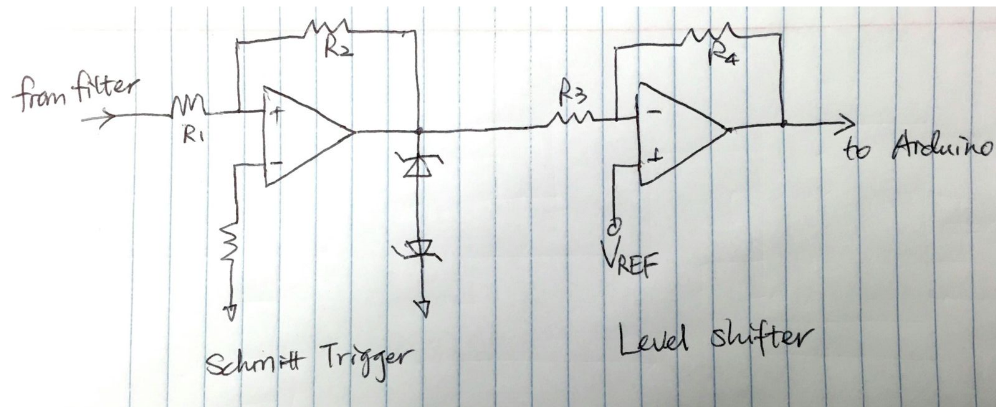


Figure 5. Schematic of analog frequency counter circuit

Microcontroller
We used an Arduino Uno as the microcontroller. The Arduino code is included in the Appendix. There are two main components in the code: 1) count the frequency of the input square wave, and 2) output a PWM signal to the servo based on the difference between the read-in frequency and the desired frequency.

The frequency is counted using an analog comparator interrupt. We compare the input square wave with 1.1V (done by setting the 6th bit of the ACSR register for the analog comparator to 1). Every time there is a rising edge, the interrupt request routine is executed and the count is incremented. This continues until 50 cycles are counted, at which point the Arduino calculates the average frequency over these 50 cycles by dividing the total time elapsed by the number of cycles. This average frequency can be affected by startup jitter, noise, etc., so frequencies more than 30Hz away from the target frequency are rejected.

The servo is driven by PWM output from the Arduino. The Arduino will only attempt to tune the string when the parsed frequency value is within the 30Hz range of the desired frequency. The Arduino will continue to sample to frequency and turn the servo until it detects a frequency that is +/- 1 Hz away from the desired frequency. Once the Arduino detects that the string is in tune, the Arduino turns a green LED on and stops all movement of the servo. The LED will remain on and the servo will not turn until the Arduino receives a new string input to tune.

DC Servo and Guitar Peg Attachment

The DC servo drives a mechanical device made of plastic, which is attached to one of the guitar pegs. When the servo rotates according to the PWM output from Arduino, it turns the peg and drives it to the desired position. The microphone is listening to the guitar note continuously, and when the system detects that the string has been tuned (i.e., the peg is turned into the correct position), the Arduino will stop the DC servo and light up the green LED.

Appendix: Arduino Code

```
/*
 * Frequency measure code adapted from code by
 * Dr. Jeffrey Groff, Assitant Professor of Physics, Shepherd University
 * http://www.fiz-ix.com/2012/11/measuring-signal-frequency-with-arduino/
 *
 * This code measures the frequency of an incoming square-wave
 * signal using timer0/micros() and the onboard comparator.
 * (It should work reqardless of the duty-cycle and for
 * non-square-wave signals as long as the signal voltage drops
 * below the bandgap voltage of the onbaord comparator (~ 1.1 V)
 * and is sufficiently clean to prevent bounding.
 */

#include <Servo.h>

// Frequency measure stuff
int inputPin = 7;
volatile unsigned long edges = 0; // rising edges counted
volatile unsigned long tstart = 0; // time when starting edge count
volatile unsigned long tstop = 0; // time when stopping edge count
volatile unsigned long tnow = 0; // temporarily stores time
const unsigned long cycles = 100; // number of cycles to avg freq over ***
float frequency = 0; // curr measured freq

// String frequencies
#define EL 82.41
#define A  110.
#define D  146.83
#define G  196.
#define B  246.94
#define EH 329.63
```

```
// Servo stuff
int string = 0; // which string to tune
int prevString = 0;
float stringFrequency = 0;
int doneTuning = 0;
int LEDPin = 2;
Servo myServo;

void setup(void) {
  // Analog comparator negative input (input signal to Arduino)
  pinMode(inputPin, INPUT);
  // Enable global interrupts, just in case (should already be done)
  SREG = SREG | B10000000;

  myServo.attach(6);
  pinMode(LEDPin, OUTPUT);

  Serial.begin(9600);
}

void loop(void) {
  delay(100);

  measureFreq();
  Serial.println(frequency);

  // Find string that needs to be tuned
  string = 0;
  for (int pin = 8; pin < 14; pin++) {
    if (digitalRead(pin) == LOW) {
      string = pin - 7;
      break;
    }
  }
  // Set target string frequency
  switch (string) {
    case 1:
      stringFrequency = EL;
      break;
    case 2:
      stringFrequency = A;
      break;
    case 3:
```

```
      stringFrequency = D;
      break;
    case 4:
      stringFrequency = G;
      break;
    case 5:
      stringFrequency = B;
      break;
    case 6:
      stringFrequency = EH;
      break;
    default:
      stringFrequency = 0;
      break;
  }
  if (string != prevString) {
    doneTuning = 0;
  }
  // Tune servo based on tuning
  if (!doneTuning && frequency - stringFrequency >= -20 && frequency - stringFrequency
<= 20) {
    tuneServo();
  }
  prevString = string;

  digitalWrite(LEDPin, doneTuning);

  delay(100);
}

void measureFreq(void) {
  edges = 0;

  // Enable analog comparator interrupt on falling edge (bit 1),
  // which would actually capture a rising edge of the signal
  // and use the internal bandgap reference voltage as the
  // positive input (bit 6).
  ACSR = ACSR | B01000010;

  delay(5); // short wait for bandgap voltage to stabilize

  // Enable analog comparator interrupt (bit 3)
  ACSR = ACSR | B00001000;
```

```
  while (edges < (cycles+1)) {
    // Do nothing
  }

  // Calculate the frequency.
  frequency = (float)1000000 * (float)cycles / (float)(tstop - tstart);
}

void tuneServo(void) {
  float diff = frequency - stringFrequency;
  if (diff < -1) { // then tune up
    Serial.println("Tuning up."); // ***
    myServo.write(180);
    delay(-20 * diff);
    myServo.writeMicroseconds(1500);
  }
  else if (diff > 1) { // then tune down
    Serial.println("Tuning down."); // ***
    myServo.write(0);
    delay(20 * diff);
    myServo.writeMicroseconds(1500);
  }
  else { // i.e. frequency is within 1 Hz
    Serial.println("Celebrate! It's tuned!"); // ***
    doneTuning = 1;
  }
}

ISR(ANALOG_COMP_vect) {
  tnow = micros(); // current time
  edges++;

  if (edges == 1) { // start counting edges
    tstart = tnow;
  }
  else if (edges == cycles + 1) { // stop counting edges
    tstop = tnow;

    // Turn off comparator
    ACSR = 0;
  }
}
```