Self-Playing Guitar

Julia Estrada, Sharjeel Laeeq, Cooper Clark

12 December 2021

1 Opportunity

Our device attempts to aid people who are physically unable to play the guitar but would like to. We aimed to create a device that could help a person strum and/or fret the guitar. This purpose for our device calls for a modular design, to allow a user to use one or both subsystems independently.

2 Strategy

The strummer is affixed to the guitar near the sound hole. The ability to move it back and forth across the strings was realized as a four-bar crank-rocker driven by a speed-controlled DC motor. The strummer speed can be changed, and it can be toggled on and off. We did not end up implementing the ability to change the strum rhythm or push to strum.

The fretter is attached to the neck of the guitar, and can play one chord (in addition to the open strings). The strings are pressed by an array of pegs on a 3D printed plate acting as a lever arm, actuated by a servo motor. The motor for the fretter is driven by a separate microcontroller than the strummer. We intended to have the fretter play at least two chords, but were not able to replace the second chord shape part in time when we discovered it interfered with the other chord shape. The controls on the fretter simply initiate an automatic toggle between the chord being applied (closed) and not applied (open), on a predefined time delay.

Ideally, the two subsystems would be in communication, but our implementation relies on the user tuning the strummer to match the speed of the fretter applying and releasing the chord shape.

3 Photos

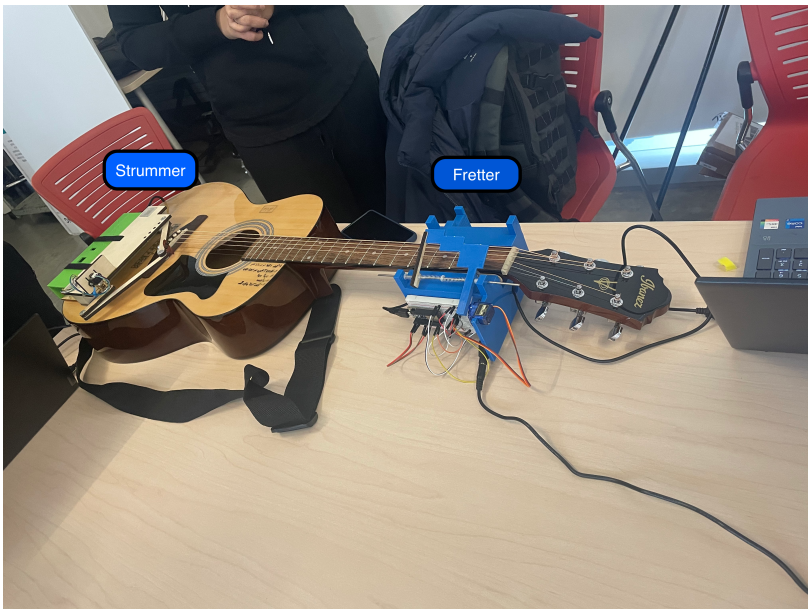


Figure 1: Device Photo

Figure 1 identifies the subsystems in our device.

4 Design Decisions and Calculations

4.1 Fretter

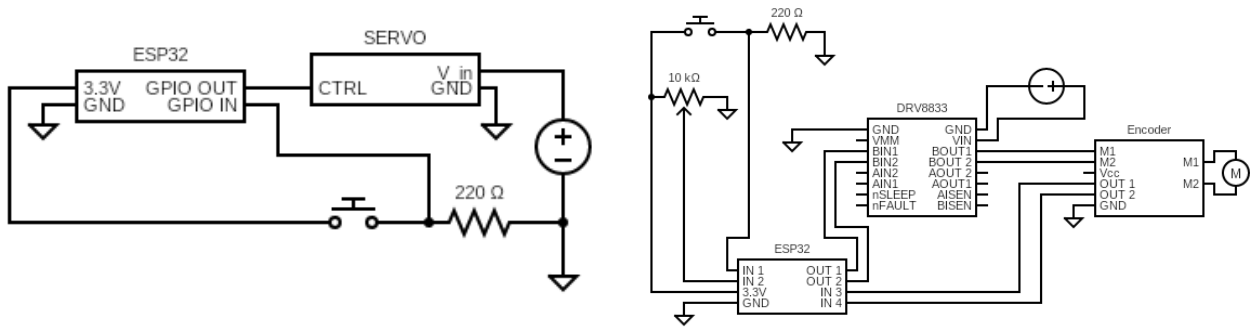
4.2 Rocker Arm

We needed to design a crank rocker such that its movement can then be used to strum the guitar. We used the following dimensions for the crank rocker. $s = 2$ in., $l = 6$ in., $p = 4$ in, $q = 6$ in., Which satisfies the equation

$$s + l < p + q$$

. To evaluate we have T1 = 6+6-2-4 = 6, T2 = 4+6-2-6 = 2, and T3 = 6+4-6-2 = 2. These calculations show that we do indeed have a crank rocker. The degree of freedom is F = 3(4-1)-2(4) = 1 and hence only one motor is required to realize the system.

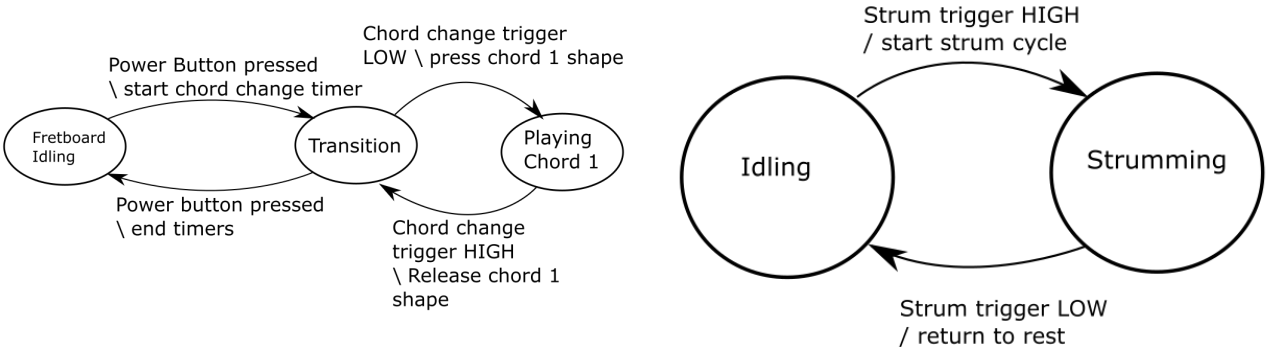
5 Circuit and State Diagrams



(a) Fretter Circuit

(b) Strummer Circuit

Figure 2: Circuits



(a) Fretter State Diagram

(b) Strummer State Diagram

Figure 3: State Diagrams

Figure 2 shows the circuit diagrams for the strummer and fretter and figure 3 shows the updated state diagrams for the two subsystems.

6 To Future Students

We wished we would have had a complete prototype with more time before the deadline. This would have enabled us to extend the functionality of our device, to better meet our ambitions when pitching the project.

Appendix A: Bill of Materials

Item No.	Qty.	List Name	Source	Cost
1	2	ESP 32 Microcontroller	Lab Kit	$0.00
2	2	5V Power Supply	Lab Kit	$0.00
3	2	5V Converter	Lab Kit	$0.00
4	1	DRV 8833 Driver	Lab Kit	$0.00
5	1	Brushed DC Motor	Lab Kit	$0.00
6	1	Motor Mounting Bracket	Lab Kit	$0.00
7	1	Collar	Lab Kit	$0.00
8	30	Jumper Wire	Lab Kit	$0.00
9	200	NEIKO 50456A Spring Assortment Set	Amazon	$11.99
10	10	Awclub 2.5x200mm 304 Stainless Steel Solid Round Rod	Amazon	$8.49
11	5	Mini Servo SG90 9g	Amazon	$10.99
12	2	Breadboard	Lab Kit	$0.00
14	500	Eowpower 500Pcs White Nylon Flat Washers	Amazon	$10.89
15	4	VELCRO Heavy Duty Fasteners 4"x2" Strips Qty. 4	Amazon	$5.79
16	4	VELCRO Strips with Adhesive 4 ct. 3 1/2 x 3/4"	Amazon	$2.98
17	100	VELCRO Cable Ties 100Pk 8 x 1/2"	Amazon	$11.49
18	4	Pin Joint	Lab Kit	$0.00

Appendix B: CAD

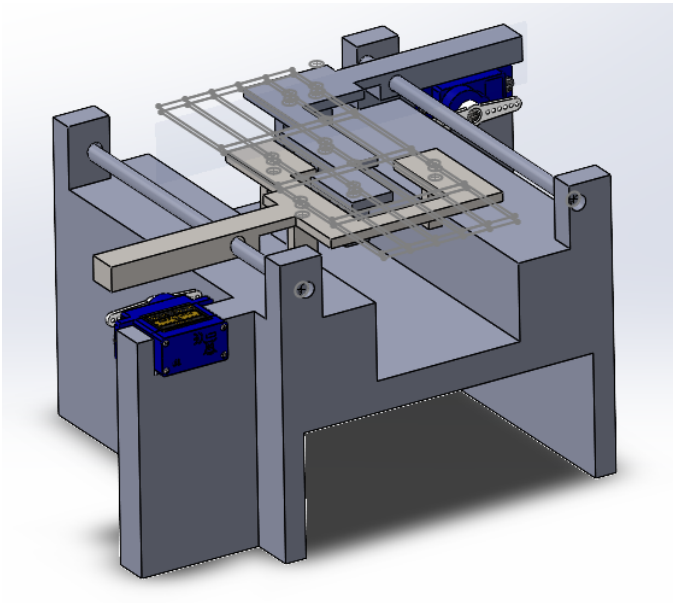


Figure 4: Fretter CAD.

Figure 4 shows an isometric view of the fretter, with two chord shapes. The G chord was not ready in time for the project showcase, but the updated design is shown. The strummer was built from lasercut parts, and thus did not require a 3D CAD file to produce, figures 5, 6, 7, 8, 9, 10 show the 2D files used for laser cutting.

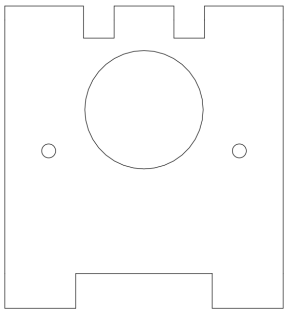


Figure 5: Strummer Part 1

Figure 6: Strummer Part 2

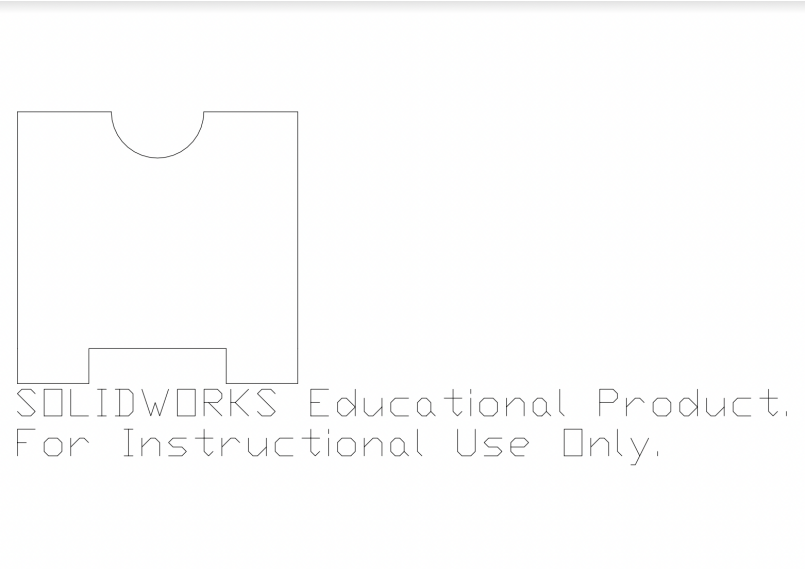


Figure 7: Strummer Part 3

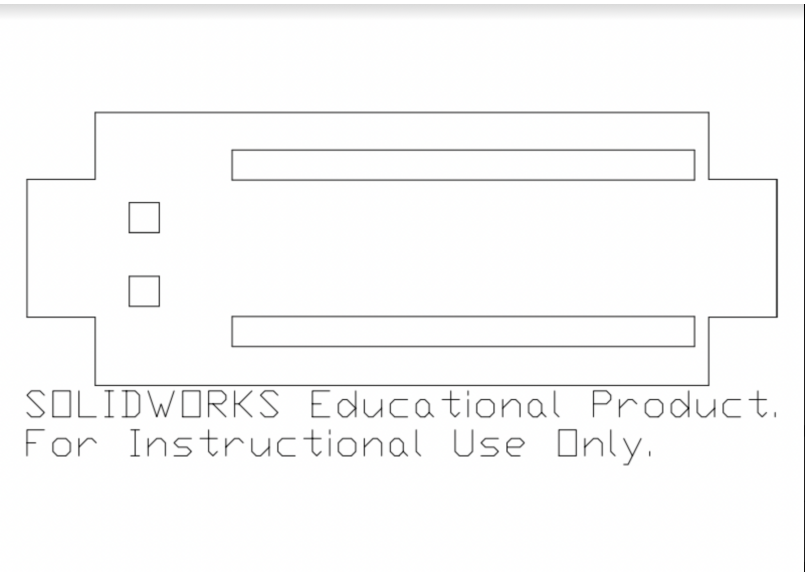


Figure 8: Strummer Part 4

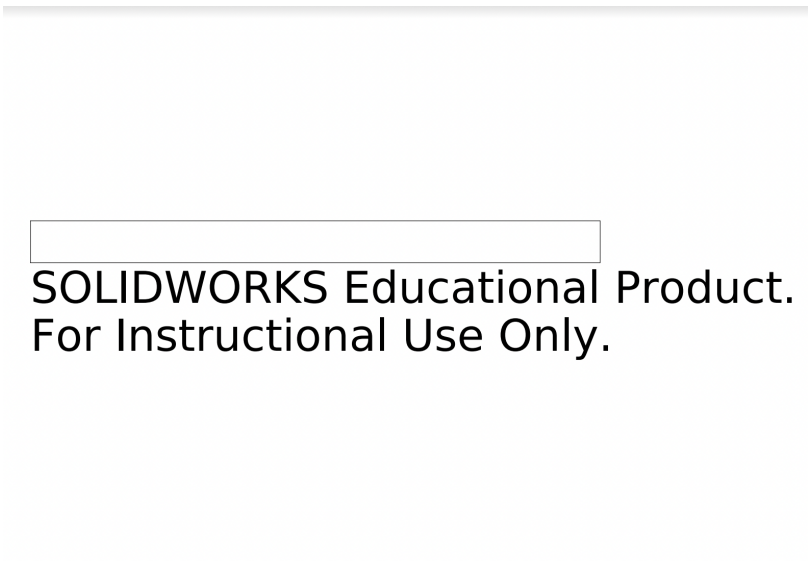


Figure 9: Strummer Part 5

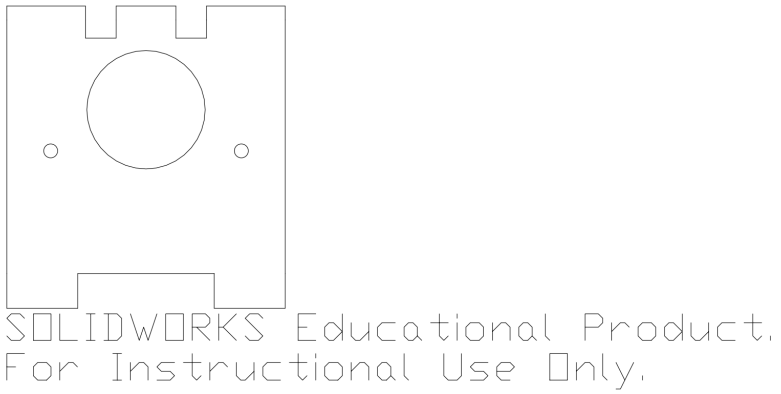


Figure 10: Strummer Part 6

Appendix C: Code

6.1 Strummer

```
#include <ESP32Encoder.h>
#define BIN_1 26
#define BIN_2 25
#define LED_PIN 13
#define BTN 12
#define POT 14

ESP32Encoder encoder;

int omegaSpeed = 0;
int omegaDes = 0;
int D = 0;
byte state = 0;
int kp = 45;
float ki = 0.01;
float te = 0;

//Setup interrupt variables ----------------------------
volatile int count = 0; // encoder count
volatile bool interruptCounter = false;    // check timer interrupt 1
volatile bool deltaT = false;     // check timer interrupt 2
int totalInterrupts = 0;   // counts the number of triggering of the alarm
hw_timer_t * timer0 = NULL;
hw_timer_t * timer1 = NULL;
portMUX_TYPE timerMux0 = portMUX_INITIALIZER_UNLOCKED;
portMUX_TYPE timerMux1 = portMUX_INITIALIZER_UNLOCKED;
volatile bool buttonIsPressed = false;

// setting PWM properties ----------------------------
const int freq = 5000;
const int ledChannel_1 = 1;
const int ledChannel_2 = 2;
const int resolution = 8;
const int MAX_PWM_VOLTAGE = 255;
const int NOM_PWM_VOLTAGE = 150;

//Initialization -------------------------------------
void IRAM_ATTR onTime0() {
  portENTER_CRITICAL_ISR(&timerMux0);
  interruptCounter = true; // the function to be called when timer interrupt is triggered
  portEXIT_CRITICAL_ISR(&timerMux0);
}

void IRAM_ATTR onTime1() {
  portENTER_CRITICAL_ISR(&timerMux1);
  count = encoder.getCount( );
  encoder.clearCount ( );
  deltaT = true; // the function to be called when timer interrupt is triggered
  portEXIT_CRITICAL_ISR(&timerMux1);
```

```
}

void IRAM_ATTR isr() {  // the function to be called when interrupt is triggered
    buttonIsPressed = true;
}

void setup() {
  // put your setup code here, to run once:
  pinMode(BTN, INPUT); // configures the specified pin to behave either as an input or an output
  pinMode(POT, INPUT);
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, LOW); // sets the initial state of LED as turned-off
  attachInterrupt(BTN, isr, RISING);

  Serial.begin(115200);
  ESP32Encoder::useInternalWeakPullResistors = UP; // Enable the weak pull up resistors
  encoder.attachHalfQuad(33, 27); // Attache pins for use as encoder pins
  encoder.setCount(0);  // set starting count value after attaching

  // configure LED PWM functionalitites
  ledcSetup(ledChannel_1, freq, resolution);
  ledcSetup(ledChannel_2, freq, resolution);

  // attach the channel to the GPIO to be controlled
  ledcAttachPin(BIN_1, ledChannel_1);
  ledcAttachPin(BIN_2, ledChannel_2);

  // initilize timer
  // timer 0, MWDT clock period = 12.5 ns * TIMGn_Tx_WDT_CLK_PRESCALE -> 12.5 ns * 80
  -> 1000 ns = 1 us, countUp
  timer0 = timerBegin(0, 80, true);
  timerAttachInterrupt(timer0, &onTime0, true); // edge (not level) triggered
  timerAlarmWrite(timer0, 2000000, true); // 2000000 * 1 us = 2 s, autoreload true

  // timer 1, MWDT clock period = 12.5 ns * TIMGn_Tx_WDT_CLK_PRESCALE -> 12.5 ns * 80
  -> 1000 ns = 1 us, countUp
  timer1 = timerBegin(1, 80, true);
  timerAttachInterrupt(timer1, &onTime1, true); // edge (not level) triggered
  timerAlarmWrite(timer1, 10000, true); // 10000 * 1 us = 10 ms, autoreload true

  // at least enable the timer alarms
  timerAlarmEnable(timer0); // enable
  timerAlarmEnable(timer1); // enable
}

void loop() {
  // put your main code here, to run repeatedly:
  switch (state) {

    case 0 : // motor is stopped
      if (buttonPressEvent()) {
        state = 1;
```

```
    startMotorResponse();
  }
  break;

case 1 : // motor running
  if (deltaT) {
    portENTER_CRITICAL(&timerMux1);
    deltaT = false;
    portEXIT_CRITICAL(&timerMux1);

    omegaSpeed = count;

    //EDIT THIS SECTION FOR A5 ASSIGNMENT
    //For A5 you will need to decide how to define omegaDes
    //It is recommended to measure the maximum speed (in counts per time loop)
    //while under no-load when NOM_PWM_VOLTAGE is applied.
    //Then, take this term and use it to compute the difference between
    //real and desired speed. Use this difference to implement
    //the various control strategies in A5.

    //Stand-in mapping between the pot reading and motor command.
    omegaDes = map(analogRead(POT), 0, 4095, -21, 21);
    float error = omegaDes-omegaSpeed;
    te = te + (omegaDes-omegaSpeed);




    D = kp*(omegaDes-omegaSpeed) + ki*te;


    //END A5 CONTROL SECTION

    //Ensure that you don't go past the maximum possible command
    if (D > MAX_PWM_VOLTAGE) {
      D = MAX_PWM_VOLTAGE;
    }
    else if (D < -MAX_PWM_VOLTAGE) {
      D = -MAX_PWM_VOLTAGE;
    }

  //Map the D value to motor directionality
  //Assumes encoder direction is same as GSI
  if (D > 0) {
    ledcWrite(ledChannel_1, LOW);
    ledcWrite(ledChannel_2, D);
  }
  else if (D < 0) {
    ledcWrite(ledChannel_2, LOW);
    ledcWrite(ledChannel_1, -D);
  }
```

```
      plotControlData();
      }

      if (buttonPressEvent()) {
      state = 0;
      stopMotorResponse();
      }
      break;

      default: // should not happen
      Serial.println("SM_ERROR");
      break;
  }
}

//Event Checkers
bool buttonPressEvent() {
  if (buttonIsPressed == true){
    buttonIsPressed = false;
    return true;
  }
  else {
    return false;
  }
}

//Event Service Responses
void stopMotorResponse() {
  ledcWrite(ledChannel_2, LOW);
  ledcWrite(ledChannel_1, LOW);
  digitalWrite(LED_PIN, LOW);
}

void startMotorResponse() {
  ledcWrite(ledChannel_1, LOW);
  ledcWrite(ledChannel_2, D);
  digitalWrite(LED_PIN, HIGH);
}

//Other functions

void plotControlData() {
  Serial.println("Speed, Desired_Speed, PWM_Duty");
  Serial.print(omegaSpeed);
  Serial.print(" ");
  Serial.print(omegaDes);
  Serial.print(" ");
  Serial.println(D/5);  //PWM is scaled by 1/10 to get more intelligible graph
}
```

6.2 Fretter

```
    #include <ESP32Servo.h>
#define LED 13
#define BTN 21
#define servoPIN 17

//SETTINGS: for easy updating
int pos = 0;    //position read by servo
int posOFF = 85;
int posON = 0;
int chTimerTics = 5000000; //for controller timer
int dbTimerTics = 25000; //for debounce timer
int servoTimerTics = 50000; //for servo timer

//init constants
byte state = 0;
int posDes = 0; //desired position
int posCmd = 0; //position motor is told to go to

//init servos
Servo servo;

//Setup interrupt variables ----------------------------
volatile bool dbTimerIsDone = false;      // check debounce timer interrupt
volatile bool chTimerIsDone = false;      // check chord change timer
volatile bool servoTimerIsDone = false;   //check servo timer
volatile bool buttonIsPressed = false;    //check button press

//timer for debouncing
hw_timer_t * dbTimer = NULL;
portMUX_TYPE dbTimer = portMUX_INITIALIZER_UNLOCKED;

//timer for chord change
hw_timer_t * chTimer = NULL;
portMUX_TYPE chTimer = portMUX_INITIALIZER_UNLOCKED;

//timer to give servos time to move
hw_timer_t * servoTimer = NULL; //timer servo delay
portMUX_TYPE servoTimer = portMUX_INITIALIZER_UNLOCKED;

//Initialization -----------------------------------
void IRAM_ATTR onTime0() {//debounce timer callback
  portENTER_CRITICAL_ISR(&dbTimer);
  dbTimerIsDone = true; // the function to be called when timer interrupt is triggered
  portEXIT_CRITICAL_ISR(&dbTimer);
}

void IRAM_ATTR onTime1() {//chord change callback
  portENTER_CRITICAL_ISR(&chTimer);
  chTimerIsDone = true;
  portEXIT_CRITICAL_ISR(&chTimer);
}
```

```
void IRAM_ATTR onTime2() {//motor delay callback
  portENTER_CRITICAL_ISR(&servoTimer);
  servoTimerIsDone = true;
  portEXIT_CRITICAL_ISR(&servoTimer);
}

void IRAM_ATTR isr() {// the function to be called when interrupt is triggered
  buttonIsPressed = true;
}

void setup() {
  // put your setup code here, to run once:
  // configure button pin
  pinMode(BTN, INPUT);
  attachInterrupt(BTN, isr, RISING);
  // configure LED pin
  pinMode(LED, OUTPUT);
  digitalWrite(LED, LOW); // sets the initial state of LED as turned-off
  // configure servo
  servo.setPeriodHertz(50);
  servo.attach(servoPIN);

  // initialize servo
  Serial.begin(115200);

  // initilize timers
  dbTimer = timerBegin(0, 80, true); //timer 0
  timerAttachInterrupt(dbTimer, &onTime0, true); // edge (not level) triggered
  timerAlarmWrite(dbTimer, dbTimerTics, true);

  chTimer = timerBegin(1, 80, true);
  timerAttachInterrupt(chTimer, &onTime1, true);
  timerAlarmWrite(chTimer, chTimerTics, true);

  servoTimer = timerBegin(2, 80, true);
  timerAttachInterrupt(servoTimer, &onTime2, true);
  timerAlarmWrite(servoTimer, servoTimerTics, true);

  // enable timers
  timerAlarmEnable(dbTimer);
  timerAlarmEnable(chTimer);
  timerAlarmEnable(servoTimer);
}

void loop() {
  // put your main code here, to run repeatedly:
  switch (state) {
      if (CheckForServoTimerDone() == false) {
        break;
      }
    case 0: //servo not running, in OFF position
```

```
      if (ButtonPressEvent()) {
        //turn on servo
        state = 2;
        digitalWrite(LED, HIGH);
        Serial.println("Enabling Fretter");
        TurnOnServo();
      }
      break;

    case 1 : // servo is in OFF position
      if (ButtonPressEvent()) {
        // disable servo (already in OFF position)
        state = 0;
        digitalWrite(LED, LOW);
        Serial.println("Disabling Fretter");
        TurnOffServo();
        break;
      }
      if (CheckForChTimerDone()) {
        //move servo to ON position
        Serial.println("Fretting");
        state = 2;
        TurnOnServo();
      }
      break;

    case 2 : // servo is in ON position
      if (ButtonPressEvent()) {
        //move servo to OFF position, disable servo
        state = 0;
        timerStop(chTimer); //disable chord change timer
        Serial.println("Disabling Fretter");
        TurnOffServo();
        digitalWrite(LED, LOW);
        break;
      }
      if (CheckForChTimerDone()) {
        //return to OFF position, wait to change again
        Serial.println("Unfretting");
        state = 1;
        TurnOffServo();
      }
      break;

    default: // should not happen
      Serial.println("SM_ERROR");
      break;
  }
}

//Event Checkers
bool ButtonPressEvent() {
```

```
  //checks if button has been pressed, with debouncing!
  if (buttonIsPressed == true) { //if button is pressed...
    if (CheckForDbTimerDone()) {//...AND timer is done
      buttonIsPressed = false; //reset
      Serial.println("Pressed!");
      return true;
    }
    else {
      return false;
    }
  }
  else if (buttonIsPressed == false) {
    return false;
  }
}

bool CheckForDbTimerDone() {
  //checks if debounce timer is done
  if (dbTimerIsDone == true) {
    dbTimerIsDone = false;
    return true;
  }
  else if (dbTimerIsDone == false) {
    return false;
  }
}

bool CheckForChTimerDone() {
  //checks if chord change timer is done
  if (chTimerIsDone == true) {
    chTimerIsDone = false;
    return true;
  }
  else if (chTimerIsDone == false) {
    return false;
  }
}

bool CheckForServoTimerDone() {
  if (servoTimerIsDone == true) {
    servoTimerIsDone = false;
    return true;
  }
  else if (servoTimerIsDone == false) {
    return false;
  }
}

//Event Service Responses

void TurnOnServo() {
  //only move servo if timer says servo is done moving
```

```
  servo.write(posON);
  timerRestart(servoTimer); //won't allow servo to move again until timer is done
  timerRestart(chTimer);
}

void TurnOffServo() {
  //only move servo if timer says servo is done moving
  servo.write(posOFF);
  timerRestart(servoTimer); //won't allow servo to move again until timer is done
  timerRestart(chTimer);
}

//Other functions

void plotControlData() {
  Serial.println("pos, posDesired, posCmd");
  Serial.print(pos);
  Serial.print(" ");
  Serial.print(posDes);
  Serial.print(" ");
  Serial.println(posCmd);
}
```