University of California, Berkeley

Boston Terrier

Quadruped Robotic Companion and Assistant

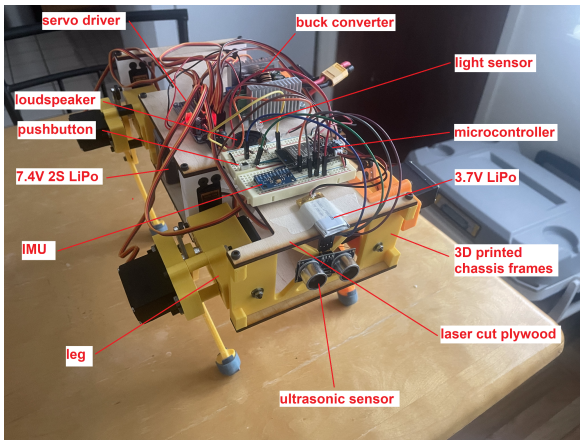
Group 26
Karen Chen, Bryant La, Moises Medrano
ME 102B: Mechatronics Design
Professor Hannah Stuart

Opportunity

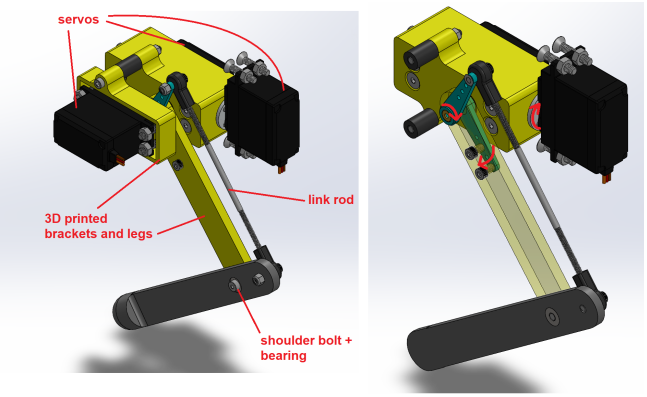
Today, robotic companions are used in many facets of daily life to provide personal companionship or complete simple tasks. For this project, our group took inspiration from recent innovations in consumer robotics, such as the Boston Dynamics dog and PARO Therapeutic Robot. We decided to pursue the design and manufacturing of the Boston Terrier, a small-scale four-legged robot that could provide companionship and personal assistance to a user.

Strategy

The Boston Terrier replicates a quadrupedal walking gait using four servo-controlled legs, with each leg having three degrees of freedom. To facilitate forward motion, the robot dog alternates between stance and swing phases by rotating and lifting its hip and knee joints. The design uses rear facing knees to maximize push-off action from the ground, resulting in faster and more efficient walking motion.



integrated device



leg subsystem

The Boston Terrier receives and follows commands sent to the microcontroller via Bluetooth; these commands include: walking forward, turning, barking, and stopping. A limitation discovered during testing is that in order to maintain its balance, the dog must stop and return to a neutral standing position before transitioning between different moving states, e.g., switching from walking to turning.

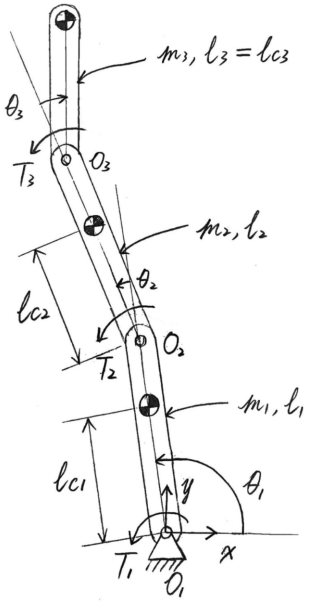
An ultrasonic range finder is used to implement obstacle avoidance by commanding the dog to turn while an object is detected within a certain threshold of the sensor. Although we removed the guard function from our original proposed strategy, we were able to implement an alert feature that is activated by an ambient light sensor. This feature will detect if the current room is too dark and play the attached loudspeaker to alert the user by “barking”. The Boston Terrier can also be made to sit down and get up by pressing a button, but some additional commands or tricks we initially desired were not implemented due to being outside the scope of the project.

Specifications

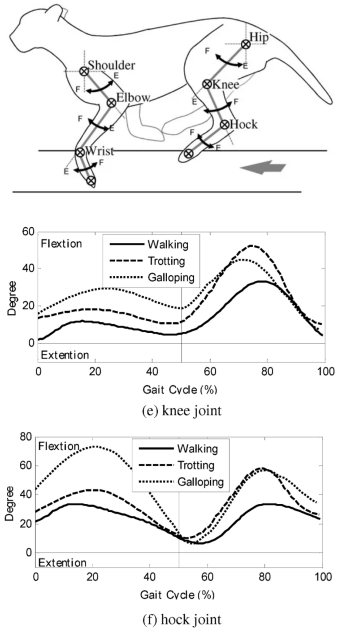
In robotic locomotion applications, the stance phase determines the minimum torque requirements for a joint motor. This is because it requires more torque to support the body weight against the ground than it does to swing the leg in the air. The leg during stance is modelled as an inverted planar triple pendulum of pinned links, with the foot removed and the ankle fixed to the ground. Lagrangian

mechanics are then used to determine the torque τ_q exerted at a joint q as a differential equation of total kinetic energy K, total potential energy P, and joint angles θ_q: $\frac{d}{dt}\left(\frac{\partial K}{\partial \dot{\theta}_q}\right) - \frac{\partial K}{\partial \theta_q} + \frac{\partial P}{\partial \theta_q} = \tau_q$.

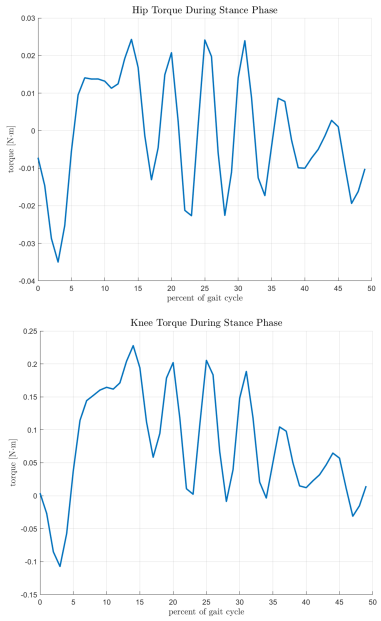
For a human walking gait cycle, the modelled knee and hip torques are larger than the torques found experimentally with torque sensors. Specifically, joint torques are overestimated by a factor of 5 due to simplifying assumptions. When applying this model for the actuator specifications of a small-scale quadrupedal walking robot, estimates were used for the mass of the chassis and legs, assuming the chassis is supported by at least three legs at all times for stability.



leg linkage model



quadruped walking

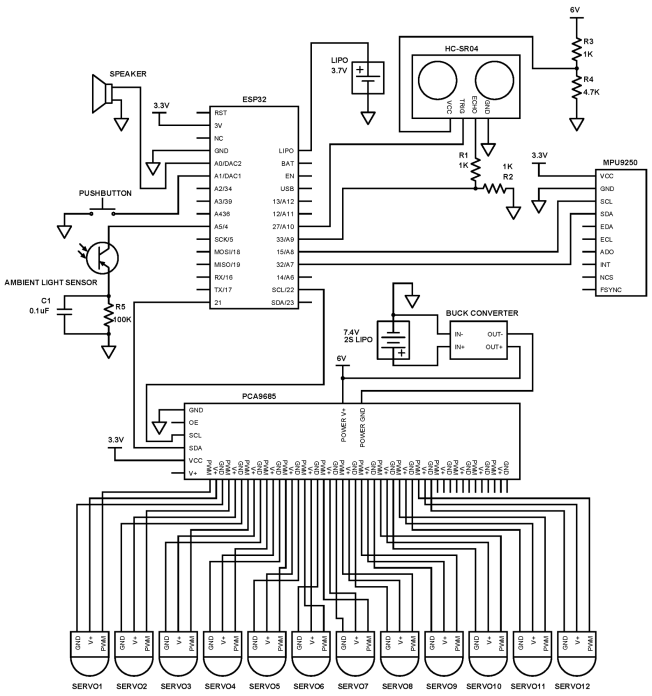


modelled joint torques

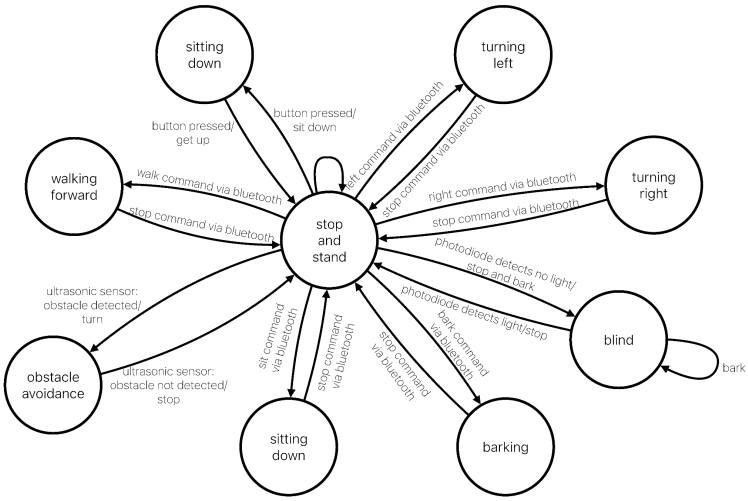
The torque requirement for the joint actuators in this project was 0.25 N·m or 2.55 kg·cm nominal torque. The results of this model were reasonable because: 1) we knew from the bipedal walking model that the modelled torque was an overestimate, and 2) other hobby roboticists online have successfully used similarly sized servos in their projects. The datasheets for most servos did not provide the maximum radial load of the servo; however, we anticipated that no radial load specifications would be exceeded given that our parts are fabricated to be as light as possible, i.e., 3D printing and laser cut wood.

For this project, we decided to use MG 996R servos. With a torque of 15.2 kg·cm at 6V and a no-load speed of 0.15s / 60 deg, the component was easily able to meet our torque requirements while providing fast position control at an attainable source voltage. Servo motors were chosen over stepper motors and DC motors with encoders to minimize weight and power consumption and simplify hardware and software.

The other electronic components (ultrasonic range finder, light sensor, pushbutton, loudspeaker, IMU, microcontroller, servo driver, buck converter, and batteries) were sourced from the MicroKit or purchased based on the electrical requirements of the servos. The functionality of some of the sensors had been previously verified through in-class lab assignments.



circuit diagram



state transition diagram

Reflection

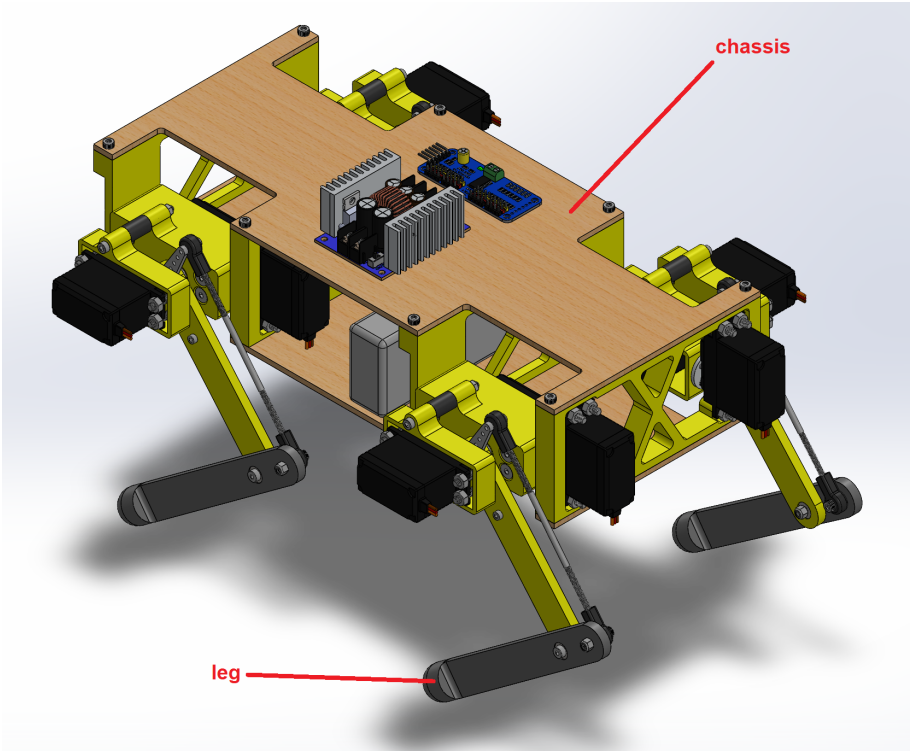
After looking back on our group's project and the final result, we have identified several aspects that we would like to improve if we could do it over again. Firstly, we would create and follow a more uniform manufacturing process for our assembly of the legs. During testing and the project showcase, it was apparent that the dog's operation was negatively impacted by the inconsistent manufacturing, such as extension rods being installed at varying lengths and certain bearings not being consistently aligned. The mechanical inconsistency between parts resulted in the legs of the robot being in slightly different positions despite receiving the same servo commands; it was difficult to reliably calibrate or compensate for these physical errors in the code. This misalignment did, however, lead our group to discover that ball joints and tie rods allowed for alignment error between parts without significant problems to overall functionality. We recommend using ball joint ends when possible for building mechanical linkages.

Another aspect of improvement we would focus on is the reduction of mechanical compliance throughout the assembly, as such compliance caused more vibration than anticipated. These vibrations required constant maintenance in order to sustain functionality (applying threadlocker to fasteners). One solution to reducing vibration would be to preload our bearings as was taught in class. Unfortunately, due to time, cost, and space constraints, this was an aspect that we had to compromise on. The reliance on 3D printing for our manufacturing was also another area of compromise. While 3D printing enables rapid fabrication times, the 3D printed PLA was too soft of a material and gradually deformed over time. This required us to super glue several ball bearings as their press fits became too loose to hold them in place. Overall, these compromises were acceptable as we were still able to deliver reasonably consistent performance for this high fidelity prototype.

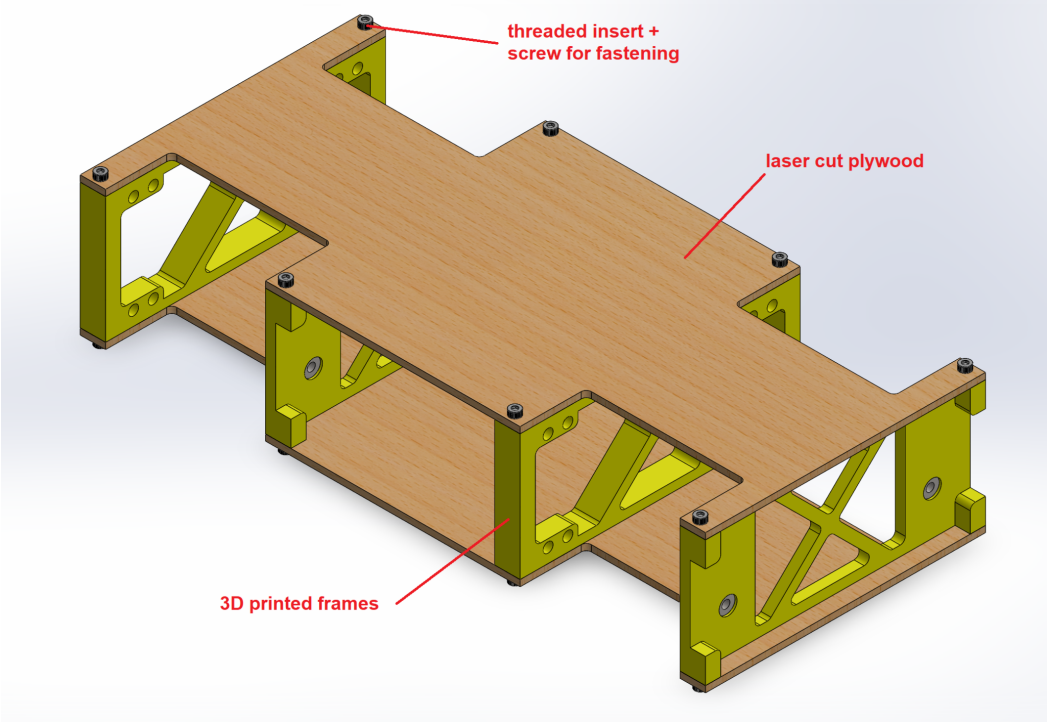
Bill of Materials (ASME Maker Grant Purchase Portfolio

Boston Terrier's						
Item Name	Description	Purchase Justification	Serial Number / SKU	Price (ea.)	Quantity	Vendor
ESP32 Microcontroller	ESP32-based microcontroller board	Used to control all of the electronics	-	$ -	1	MicroKit
HC-SR04 Ultrasonic Sensor	Ultrasonic sonar distance sensor	Used to detect distance to objects	-	$ -	1	MicroKit
Mini Loudspeaker	1W 8Ω round internal magnet mini loudspeaker	Used to emit sounds from the robot	-	$ -	1	MicroKit
MPU9250 IMU	9-Axis 9 DOF intertial measurement unit	Used to measure acceleration, angular velocity, acceleration	-	$ -	1	MicroKit
3.7V LiPo Battery	3.7V 290mAh lithium polymer battery	Used to power the ESP32 without a USB connection	-	$ -	1	MicroKit
5V Power Supply and Connector	5V 2A switching power supply and female DC power jack adapter	For testing one servo at a time	-	$ -	1	MicroKit
MG996R Servo Motors	6V 15.2kg·cm servo motors	Used to rotate links to precise angular positions	B08CH2SJLR	$ 27.99	2	Amazon
PCA9685 Servo Motor Driver Board	16-Ch 12-Bit PWM servo motor driver board controller	Used to drive servos separately from a a 3.3V microcontroller	B07WS5XY63	$ 13.79	1	Amazon
DC-DC Buck Converter Step Down Module	DC 6V-40V to 1.2V-36V 20A 300W adjustable step down voltage regulator power supply module	Used to step down the power supply voltage to a lower voltage for the servos	B09DV6MW4Q	$ 14.99	1	Amazon
7.4V 2S LiPo Battery	5200mAh 70C 7.4V lithium polymer battery	Used to power the servos	B08L7PLHJN	$ 32.99	1	Amazon
LiPo Battery Connectors	XT60 14AWG plug connector female and male	Used to connect the battery to voltage rails	B085HF31K7	$ 10.99	1	Amazon
Servo Discs	25T aluminum servo discs, M3 threads (comes with mounting M3 screws)	Used to mount mechanisms to the servos	B07D56FVK5	$ 11.99	1	Amazon
Servo Arms	25T aluminum servo arms, M3 threads	Used to mount mechanisms to the servos	B08RHMQX7Q	$ 8.88	3	Amazon
M3 Screws	M3 x 0.5mm thread, 8mm long	Used to fasten parts together	91290A113	$ 8.36	1	McMaster Carr
M2.5 Screws	M2.5 x 0.45mm thread, 8mm long	Used to fasten parts together	91290A102	$ 9.96	1	McMaster Carr
M3 Screws, Long	M3 x 0.5mm thread, 45mm long	Used to fasten upper and lower leg servos across spacers	91290A079	$ 4.57	1	McMaster Carr
M4 Screws, Flat	M4 x 0.70mm thread, 20mm long, 90 degree countersink	Used to mount servos to brackets	91294A196	$ 8.93	1	McMaster Carr
M4 Nuts	Class 8, M4 x 0.7mm thread, steel hex nuts	Used to fasten parts together	90592A090	$ 1.41	1	McMaster Carr
4mm Ball Bearings	4mm x 9mm x 4mm double shielded carbon steel ball bearings	Used to constrain shaft rotation and reduce friction at joints	B093W5RGGC	$ 9.99	1	Amazon
M3 Ball Head Tie Rod Ends	M3 3.0 x L20mm lever steering linkage ball head tie rod ends (comes with M3 screws)	Used to mechanically connect a linkage to rotate the lower legs	B07Q2WPD2T	$ 9.99	1	Amazon
M3 Threaded Rods	M3 x 100mm 304 stainless steel partially threaded rods	Used as a push rod or linkage to rotate the lower legs	B01LXD5KPG	$ 9.99	1	Amazon
M3 Shoulder Bolts	M3, 4mm diam, 10mm length, shoulder bolt	Used as a pin joint for the knee	92981A742	$ 2.48	8	McMaster Carr
M3 Locknuts	M3 x 0.5mm thread, 4mm high, locknut	Used to fasten the shoulder bolt	90576A102	$ 4.65	1	McMaster Carr
M3 Threaded Inserts	M3 x 0.5mm thread, 5.56mm long, heat-set threaded insert	Used to fasten bolts into holes without a nut	97164A113	$ 6.30	1	McMaster Carr
1/8" Plywood	1/8" x 18" x 30" plywood	Used to fabricate the chasiss	-	$ 2.22	1	Jacobs
Servo Discs	25T aluminum servo discs, M3 threads (comes with mounting M3 screws)	Repurchasing because the previous order was misplaced/lost	B07D56FVK5	$ 11.99	1	Amazon

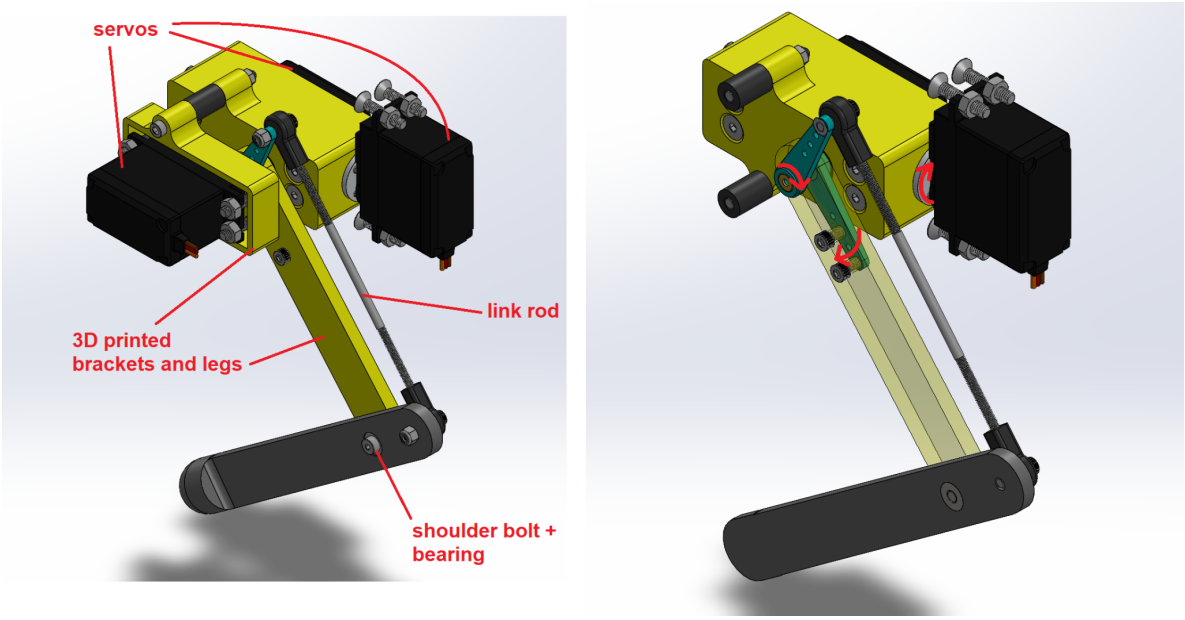
CAD Models



Full Assembly



Chassis


servos
link rod
3D printed
brackets and legs
shoulder bolt +
bearing

Leg

Code

```
/* state machine and Bluetooth control */
int state = 0;
char command = 'S';

// https://randomnerdtutorials.com/arduino-control-2-dc-motors-via-bluetooth/
// https://randomnerdtutorials.com/esp32-bluetooth-classic-arduino-ide/
#include "BluetoothSerial.h"

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

// must transmit no line ending to ESP32 in the Bluetooth Serial Terminal app
BluetoothSerial SerialBT;

/* servo driver */
// https://learn.adafruit.com/16-channel-pwm-servo-driver/hooking-it-up
// https://robojax.com/node/1193
// https://www.youtube.com/watch?v=y8X9X10Tn1k&ab_channel=Robojax
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h> // Click here to get the library: http://librarymanager/All#Adafruit_PWM_Servo_Driver_Library

#define SCL_1 22 // default SCL
#define SDA_1 21 // default SDA
Adafruit_PWMServoDriver servo;
const int servoFreq = 50; // MG996R servos run at ~50 Hz updates
const int servoMin = 75;  // minimum pulse length count (out of 4096)
const int servoMax = 500; // maximum pulse length count (out of 4096)

int rotation[16] = {1,1,1,0,  1,1,1,0,  -1,-1,-1,0,  -1,-1,-1,0}; // 1 = normal, -1 = reversed
int angleZero[16] = {105,10,12,0,  115,12,22,0,  115,195,187,0,  120,195,197,0}; // servo's zero degree position

/* movement function variables */
// RR = rear right; FR = front right; RL = rear left; FL = FL
float initPos11 =         0; // RR hip roll
float initPos12 =        35; // RR hip pitch
float initPos13 =         8; // RR knee
float initPos21 = initPos11; // FR hip roll
float initPos22 = initPos12; // FR hip pitch
float initPos23 = initPos13; // FR knee
float initPos31 = initPos11; // RL hip roll
float initPos32 = initPos12; // RL hip pitch
float initPos33 = initPos13; // RL knee
float initPos41 = initPos11; // FL hip roll
float initPos42 = initPos12; // FL hip pitch
float initPos43 = initPos13; // FL knee

float currPos11, prevPos11, targPos11; // RR hip roll
float currPos12, prevPos12, targPos12; // RR hip pitch
float currPos13, prevPos13, targPos13; // RR knee
float currPos21, prevPos21, targPos21; // FR hip roll
float currPos22, prevPos22, targPos22; // FR hip pitch
float currPos23, prevPos23, targPos23; // FR knee
float currPos31, prevPos31, targPos31; // RL hip roll
float currPos32, prevPos32, targPos32; // RL hip pitch
float currPos33, prevPos33, targPos33; // RL knee
float currPos41, prevPos41, targPos41; // FL hip roll
float currPos42, prevPos42, targPos42; // FL hip pitch
float currPos43, prevPos43, targPos43; // FL knee

byte i;   // increment() counter
byte ird; // increment rate divider
byte phase = 1;
byte phaseDelay = 50;
```

```
bool wakeUpDone = false; // to indicate if wakeUp() has reached its target positions
bool moving = false;     // indicates if it is running a motion function

/* IMU */
// http://www.esp32learning.com/code/esp32-and-mpu-9250-sensor-example.php
// https://github.com/asukiaaa/MPU9250_asukiaaa/blob/master/examples/GetData/GetData.ino
#include <MPU9250_asukiaaa.h> // Click here to get the library: http://librarymanager/All#MPU9250_asukiaaa
#define SCL_2 15 // A8
#define SDA_2 32 // A7
MPU9250_asukiaaa IMU;
// measurements (accelerometer [g], gyroscope [deg/s])
float aX, aY, aZ, gX, gY, gZ, pitch, roll;
const float g = 9.81;
// calibration
const float aX_offset = 0.32;
const float aY_offset = -0.23;
const float aZ_offset = 0.47;
const float gX_offset = 0.65;
const float gY_offset = 0.8;
const float gZ_offset = 0.95;

/* ultrasonic sensor */
// https://randomnerdtutorials.com/esp32-hc-sr04-ultrasonic-arduino/
// https://arduino.stackexchange.com/questions/36875/hc-sr04-distance-measuring-without-delay
#define TRIG 27 // A10
#define ECHO 33 // A9
float duration, distance;
int trigState;
unsigned long lastObstacleCheckTime;
const float distLowThreshold = 50; // cm
const float distHighThreshold = 55; // cm
bool obstacleFlag = false;

/* light sensor */
// https://www.electronics-tutorials.ws/io/io_4.html
// https://www.dfrobot.com/blog-611.html
// https://randomnerdtutorials.com/esp32-adc-analog-read-arduino-ide/
#define PHS 4 // A5
int photoVal;
const int photoLowThreshold = 200;
const int photoHighThreshold = 300;
bool lightOffFlag = false;

/* speaker */
// https://randomnerdtutorials.com/esp32-pwm-arduino-ide/
// https://github.com/espressif/arduino-esp32/blob/a4305284d085caeddd1190d141710fb6f1c6cbe1/cores/esp32/esp32-hal-ledc.h
#define LED_BUILTIN 13 // A12
#define SPK 26 // A0
const int SPK_freq = 5000;
const int SPK_pwmChannel = 0;
const int SPK_resolution = 8;
const int barkLength = 130; // ms
int barkDuration[] = {1,1,1,1,3,3,6,1,1,1,1,3,3,3,1,2,1,1,1,1,3,3,3,1,2,2,2,4,8,1,1,
  1,1,3,3,6,1,1,1,1,3,3,3,1,2,1,1,1,1,3,3,3,1,2,2,2,4,8,4,2*1,2*1,2*1,2*1,2*1,2*2,2*1,
  2*1,2*1,2*5,1,2*1,2*1,2*1,2*1,2*3,2*1,2*2,2*1,2*1,2*5,1,2*1,2*1,2*1,2*1,2*1,2*1,2*2,2*1,
  2*1,2*1,2*1,2*3,1,2*1,2*1,2*1,2*2,2*1,2*1,2*1,2*1,2*1,2*1,2*1,2*4,5,2*1,2*1,2*1,2*1,2*1,
  2*1,2*1,2*1,2*1,2*2,2*2,2*1,2*1,2*1,2*1,2*3,2*1,2*1,2*1,2*1,2*3};
//int bach[] = {262,330,392,523,659,392,523,659,262,330,392,523,659,392,523,659,
//  262,294,392,587,698,392,587,698,262,294,392,587,698,392,587,698,247,294,392,587,
//  698,392,587,698,247,294,392,587,698,392,587,698,262,330,392,523,659,392,523,659,
//  262,330,392,523,659,392,523,659,262,330,440,659,880,440,659,440,262,330,440,659,
//  880,440,659,440,262,294,370,440,587,370,440,587,262,294,370,440,587,370,440,587,
//  247,294,392,587,784,392,587,784,247,294,392,587,784,392,587,784,247,262,330,392,
//  523,330,392,523,247,262,330,392,523,330,392,523,247,262,330,392,523,330,392,523,
//  247,262,330,392,523,330,392,523,220,262,330,392,523,330,392,523,220,262,330,392,
```

```
//  523,330,392,523,147,220,294,370,523,294,370,523,147,220,294,370,523,294,370,523,
//  196,247,294,392,494,294,392,494,196,247,294,392,494,294,392,494}; // Bach Prelude in C
int freqArray[] = {466,466,415,415,698,698,622,466,466,415,415,622,622,554,523,466,554,554,
  554,554,554,622,523,466,415,415,415,622,554,466,466,415,415,698,698,622,466,466,415,415,
  831,523,554,523,466,554,554,554,554,554,622,523,466,415,-1,415,622,554,-1,277,277, 277,277,
  311,-1,261,233,208,-1,233,233,261,277,208,415,415,311,-1,233,233,261,277,233,277,311,-1,261,
  233,233,208,-1,233,233,261,277,208,208,311,311,311,349,311,277,311,349,277,311,311,311,349,
  311,208,-1,233,261,277,208,-1,311,349,311}; // Rickroll: https://create.arduino.cc/projecthub/410027/rickroll-piezo-buzzer-a1cd11
int freqIndex = 0;
int freqLength = sizeof(freqArray) / sizeof(int);

/* button */
#define BTN 25 // A1
const int debounceTime = 250; // ms
volatile bool buttonPressFlag = false;

void IRAM_ATTR isr() {
  static unsigned long lastInterruptTime = 0;
  unsigned long interruptTime = millis();

  // if interrupts come faster than [debounceTime] ms, assume it's a bounce and ignore
  if (interruptTime - lastInterruptTime > debounceTime) {
    buttonPressFlag = true;
  }
  lastInterruptTime = interruptTime;
}

void setup() {
  /* initialize pin modes */
  pinMode(TRIG, OUTPUT);
  pinMode(ECHO, INPUT);
  pinMode(PHS, INPUT); // HIGH when light, LOW when dark
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(SPK, OUTPUT);
  ledcSetup(SPK_pwmChannel, SPK_freq, SPK_resolution);
  ledcAttachPin(SPK, SPK_pwmChannel);
  pinMode(BTN, INPUT_PULLUP); // HIGH when open, LOW when pressed
  attachInterrupt(BTN, isr, FALLING);

  /* initialize I2C communication */
  // create I2C buses on defined pins
  Wire.begin(SDA_1, SCL_1);
  Wire1.begin(SDA_2, SCL_2);
  // initialize servo driver, I2C address = 0x40
  servo = Adafruit_PWMServoDriver(0x40, Wire);
  servo.begin();
  servo.setPWMFreq(servoFreq);
  // initialize IMU
  IMU.setWire(&Wire1);
  IMU.beginAccel();
  IMU.beginGyro();

  /* initialize serial and Bluetooth communication */
  Serial.begin(115200);
  SerialBT.begin("ESP32"); //Bluetooth device name
  Serial.println("The device started, now you can pair it with Bluetooth!");

  wakeUp();
}

void loop() {
  getBTCommand();

  /* state machine */
  switch (state) {
```

```
    case 0: // stopped, waiting for controller or sensor input
      if (command != 'S') {
        switch (command) {
          case 'F':   // walk forward
            state = 1;
            break;
          case 'L':   // turn left
            state = 2;
            break;
          case 'R':   // turn right
            state = 3;
            break;
//          case 'D':   // sit down
//            state = 4;
//            sit();
//            break;
          case 'B':   // bark
            state = 5;
            break;
          default:    // unexpected command
            Stop();
            break;
        }
      } else {
        if (checkForObstacle() == true) {
          state = 6;
        } else if (checkForLightOff() == true) {
          state = 7;
        } else if (checkForButtonPress() == true) {
          state = 8;
          sit();
        } else {
          // do nothing (remain in stopped/standing position)
        }
      }
      break;
    case 1: // walking forward
      if (command == 'S') {
        state = 0;
        Stop();
      } else {
        trot_F();
      }
      break;
    case 2: // turning left
      if (command == 'S') {
        state = 0;
        Stop();
      } else {
        turn_L();
      }
      break;
    case 3: // turning right
      if (command == 'S') {
        state = 0;
        Stop();
      } else {
        turn_R();
      }
      break;
//    case 4: // sitting down
//      if (command == 'S') {
//        state = 0;
//        Stop();
//      }
```

```
//      break;
    case 5: // barking
      if (command == 'S') {
        state = 0;
        stopBark();
        Stop();
      } else {
        startBark();
      }
      break;
    case 6: // obstacle avoidance
      if (checkForObstacle() == true) {
        turn_R();
      } else {
        state = 0;
        Stop();
      }
      break;
    case 7: // blinded
      if (checkForLightOff() == true) {
        startBark();
      } else {
        state = 0;
        stopBark();
        Stop();
      }
      break;
    case 8: // sitting down
      if (checkForButtonPress() == true) {
        state = 0;
        Stop();
      }
      break;
    default:
      Serial.println("error state");
      break;
  }
}

/* helper functions */
void getBTCommand() {
  if (SerialBT.available()) {
    command = SerialBT.read();
  }
}

// retrieves command from serial monitor
void getCommand() {
  if (Serial.available() > 0) {
    String serialCommand = Serial.readStringUntil('\n');
    char temp = Serial.read(); // remove invisible characters left in buffer
  } else {
    String serialCommand = "none";
  }
}

// maps angle in degrees to servo pulse width
int angleToPulse(int ang){
   return map(ang, 0, 180, servoMin, servoMax);
}

// sets servo on selected pin to given degree
void setServo(uint8_t pin, float angle) {
  angle = angleZero[pin] + angle*rotation[pin];
  int pulse = angleToPulse(angle);
```

```
  if (pulse > 540) {
    servo.setPWM(pin, 0, 540);
  } else if (pulse < servoMin) {
    servo.setPWM(pin, 0, servoMin);
  } else {
    servo.setPWM(pin, 0, pulse);
  }
}

/* event checker functions */
// checks if dog is off-balance
bool checkForOffBalance() {
  // update IMU readings (acceleration [m/s2], angular velocity [rad/s])
  IMU.accelUpdate();
  aX = -(IMU.accelY() - aY_offset) * g;
  aY = (IMU.accelX() - aX_offset) * g;
  aZ = (IMU.accelZ() - aZ_offset) * g;
  IMU.gyroUpdate();
  gX = (IMU.gyroX() - gX_offset) * DEG_TO_RAD;
  gY = (IMU.gyroY() - gY_offset) * DEG_TO_RAD;
  gZ = (IMU.gyroZ() - gZ_offset) * DEG_TO_RAD;

  // tilt angles: https://wiki.dfrobot.com/How_to_Use_a_Three-Axis_Accelerometer_for_Tilt_Sensing
  pitch = atan2(-aX, sqrt(aY*aY + aZ*aZ)) * RAD_TO_DEG;
  roll = atan2(aY , aZ) * RAD_TO_DEG;

  // off-balance if tilted
  if ((abs(pitch) > 10) || (abs(roll - 180) > 10)) {
    return true;
  }
  return false;
}

// checks if obstacle is too close
bool checkForObstacle() {
  static unsigned long prevTime = 0;
  unsigned long currTime = micros();

  if ((trigState == 0) && (currTime - prevTime > 2)) {
    digitalWrite(TRIG, HIGH);
    trigState = 1;
    prevTime = currTime;
  }
  if ((trigState = 1) && (currTime - prevTime > 10)) {
    digitalWrite(TRIG, LOW);
    trigState = 0;
    prevTime = currTime;

    duration = pulseIn(ECHO, HIGH);
    distance = (duration*.0343)/2;
  }

  if (distance < distLowThreshold) {
    obstacleFlag = true;
  } else if (distance > distHighThreshold) {
    obstacleFlag = false;
  }
//  Serial.println(distance);
  return obstacleFlag;
}

// checks if dog is blinded
bool checkForLightOff() {
  photoVal = analogRead(PHS);
//  Serial.println(photoVal);
```

```
  if (photoVal < photoLowThreshold) {
    lightOffFlag = true;
  } else if (photoVal > photoHighThreshold) {
    lightOffFlag = false;
  }
  return lightOffFlag;
}

// checks if button has been pressed
bool checkForButtonPress() {
  if (buttonPressFlag == true) {
    buttonPressFlag = false;
    return true;
  }
  return false;
}

/* service response functions */
// plays loudspeaker
void startBark() {
  static unsigned long lastBarkTime = 0;
  unsigned long barkTime = millis();

  // update the frequency being played every [barkLength * barkDuration] ms
  if (barkTime - lastBarkTime > barkLength * barkDuration[freqIndex]) {
    freqIndex = (freqIndex + 1) % freqLength;
    lastBarkTime = barkTime;
    ledcWriteTone(SPK_pwmChannel, freqArray[freqIndex]);
  }
  // turn LED on
  digitalWrite(LED_BUILTIN, HIGH);
}

// stops loudspeaker
void stopBark() {
  // turn speaker off
  ledcWriteTone(SPK_pwmChannel, 0);
  // turn LED off
  digitalWrite(LED_BUILTIN, LOW);
}

/* service response motion functions */
// sends PWM commands to move joints into neutral standing position when dog is first turned on
void wakeUp() {
  // RR leg                // FR leg                // RL leg                     // FL leg
  currPos11 = initPos11;   currPos21 = initPos21;   currPos31 = initPos31;        currPos41 = initPos41;
  currPos12 = initPos12;   currPos22 = initPos22;   currPos32 = initPos32 - 15;   currPos42 = initPos42;
  currPos13 = initPos13;   currPos23 = initPos23;   currPos33 = initPos33 - 10;   currPos43 = initPos43;

  updatePos();
  setPrevPos();
  delay(2000);
  wakeUpDone = true;
}

// trot angles
int a = -4; // tilt on left side to correct drift
int tU1 =  0;   int tF1 =  0;  int tD1 =  0;  int tB1 =   0;
int tU2 = 42;   int tF2 = 26;  int tD2 = 35;  int tB2 =  42;
int tU3 = 32;   int tF3 = 15;  int tD3 =  8;  int tB3 = -1;

// sends PWM commands to servos to move joints through the preprogrammed stages of the gait cycle
void trot_F() {
  Serial.println("Dog is walking.");
  moving = true;
```

```
ird = 8; // define in how many increments each phase will be divided (the higher the ird, the slower the movement)

switch (phase) {
  case 1:
    // target positions for phase 1
    // RR leg UP        // FR leg DOWN      // RL leg DOWN                // FL leg UP
    targPos11 =  tU1;   targPos21 =  tD1;   targPos31 = targPos21 + a;    targPos41 = targPos11 + a;
    targPos12 =  tU2;   targPos22 =  tD2;   targPos32 = targPos22 - 5;    targPos42 = targPos12;
    targPos13 =  tU3;   targPos23 =  tD3;   targPos33 = targPos23 - 5;    targPos43 = targPos13;

    if (i < ird) {
      increment();       // run increment function until target positions get reached
      i++;
    } else {
      delay(phaseDelay); // wait a little for it to get there
      setPrevPos();      // store current positions as previous positions for the next function
      phase = 2;         // and proceed to phase 2
      i = 0;             // reset i for the next phase
    }

    break;
  case 2:
    // target positions for phase 2
    // RR leg FRONT     // FR leg BACK      // RL leg BACK                // FL leg FRONT
    targPos11 =  tF1;   targPos21 =  tB1;   targPos31 = targPos21 + a;    targPos41 = targPos11 + a;
    targPos12 =  tF2;   targPos22 =  tB2;   targPos32 = targPos22 - 5;    targPos42 = targPos12;
    targPos13 =  tF3;   targPos23 =  tB3;   targPos33 = targPos23 - 5;    targPos43 = targPos13;

    if (i < ird) {
      increment();
      i++;
    } else {
      delay(phaseDelay);
      setPrevPos();
      phase = 3;
      i = 0;
    }

    break;
  case 3:
    // RR leg DOWN      // FR leg UP        // RL leg UP                // FL leg DOWN
    targPos11 =  tD1;   targPos21 =  tU1;   targPos31 = targPos21 + a;  targPos41 = targPos11 + a;
    targPos12 =  tD2;   targPos22 =  tU2;   targPos32 = targPos22;      targPos42 = targPos12;
    targPos13 =  tD3;   targPos23 =  tU3;   targPos33 = targPos23;      targPos43 = targPos13;

    if (i < ird) {
      increment();
      i++;
    } else {
      delay(phaseDelay);
      setPrevPos();
      phase = 4;
      i = 0;
    }

    break;
  case 4:
    // RR leg BACK      // FR leg FRONT     // RL leg FRONT               // FL leg BACK
    targPos11 =  tB1;   targPos21 =  tF1;   targPos31 = targPos21 + a;    targPos41 = targPos11 + a;
    targPos12 =  tB2;   targPos22 =  tF2;   targPos32 = targPos22 - 5;    targPos42 = targPos12;
    targPos13 =  tB3;   targPos23 =  tF3;   targPos33 = targPos23 - 5;    targPos43 = targPos13;

    if (i < ird) {
      increment();
      i++;
```

```
      } else {
        delay(phaseDelay);
        setPrevPos(); // if last phase reach its target positions...
        phase = 1;    // ... return to phase 1
        i = 0;
      }

      break;
  }

  updatePos();
}

// right turn angles
int rD1 =  0;   int rO1 =  5;   int rU1 =  0;   int rI1 =  -5;
int rD2 = 35;   int rO2 = 35;   int rU2 = 47;   int rI2 =  35;
int rD3 =  8;   int rO3 =  8;   int rU3 = 28;   int rI3 =   8;

// sends PWM commands to move hip joints inward and outward to make dog turn right
void turn_R() {
  Serial.println("Dog is turning right.");
  moving = true;
  ird = 4;

  switch (phase) {
    case 1:
      // RR leg DOWN      // FR leg UP        // RL leg UP             // FL leg DOWN
      targPos11 =  rD1;   targPos21 =  rU1;   targPos31 = targPos21;   targPos41 = targPos11;
      targPos12 =  rD2;   targPos22 =  rU2;   targPos32 = targPos22;   targPos42 = targPos12;
      targPos13 =  rD3;   targPos23 =  rU3;   targPos33 = targPos23;   targPos43 = targPos13;

      if (i < ird) {
        increment();
        i++;
      } else {
        setPrevPos();
        delay(phaseDelay);
        phase = 2;
        i = 0;
      }

      break;
    case 2:
      // RR leg OUT       // FR leg OUT       // RL leg OUT            // FL leg OUT
      targPos11 =  rO1;   targPos21 =  rO1;   targPos31 = targPos21;   targPos41 = targPos11;
      targPos12 =  rO2;   targPos22 =  rO2;   targPos32 = targPos22;   targPos42 = targPos12;
      targPos13 =  rO3;   targPos23 =  rO3;   targPos33 = targPos23;   targPos43 = targPos13;

      if (i < ird) {
        increment();
        i++;
      } else {
        setPrevPos();
        delay(phaseDelay);
        phase = 3;
        i = 0;
      }

      break;
    case 3:
      // RR leg UP        // FR leg DOWN      // RL leg DOWN               // FL leg UP
      targPos11 =  rU1;   targPos21 =  rD1;   targPos31 = targPos21;       targPos41 = targPos11;
      targPos12 =  rU2;   targPos22 =  rD2;   targPos32 = targPos22 - 15;   targPos42 = targPos12;
      targPos13 =  rU3;   targPos23 =  rD3;   targPos33 = targPos23 - 10;   targPos43 = targPos13;
```

```
      if (i < ird) {
        increment();
        i++;
      } else {
        setPrevPos();
        delay(phaseDelay);
        phase = 3;
        i = 0;
      }

      break;
    case 3:
      // RR leg UP        // FR leg DOWN      // RL leg DOWN                // FL leg UP
      targPos11 =  rU1;   targPos21 =  rD1;   targPos31 = targPos21;        targPos41 = targPos11;
      targPos12 =  rU2;   targPos22 =  rD2;   targPos32 = targPos22 - 15;   targPos42 = targPos12;
      targPos13 =  rU3;   targPos23 =  rD3;   targPos33 = targPos23 - 10;   targPos43 = targPos13;

      if (i < ird) {
        increment();
        i++;
      } else {
        setPrevPos();
        delay(phaseDelay);
        phase = 4;
        i = 0;
      }

      break;
    case 4:
      // RR leg IN        // FR leg IN        // RL leg IN             // FL leg IN
      targPos11 =  rI1;   targPos21 =  rI1;   targPos31 = targPos21;   targPos41 = targPos11;
      targPos12 =  rI2;   targPos22 =  rI2;   targPos32 = targPos22;   targPos42 = targPos12;
      targPos13 =  rI3;   targPos23 =  rI3;   targPos33 = targPos23;   targPos43 = targPos13;

      if (i < ird) {
        increment();
        i++;
      } else {
        setPrevPos();
        delay(phaseDelay);
        phase = 1;
        i = 0;
      }

      break;
  }

  updatePos();
}

// left turn angles
int lU1 =  0;   int lO1 =  5;   int lD1 =  0;   int lI1 =  -5;
int lU2 = 47;   int lO2 = 35;   int lD2 = 35;   int lI2 =  35;
int lU3 = 28;   int lO3 =  8;   int lD3 =  8;   int lI3 =   8;

// sends PWM commands to move hip joints inward and outward to make dog turn left
void turn_L() {
  Serial.println("Dog is turning left.");
  moving = true;
  ird = 4;

  switch (phase) {
    case 1:
      // RR leg UP        // FR leg DOWN      // RL leg DOWN                // FL leg UP
      targPos11 =  lU1;   targPos21 =  lD1;   targPos31 = targPos21;        targPos41 = targPos11;
```

```
  targPos12 =  lU2;   targPos22 =  lD2;   targPos32 = targPos22 - 15;   targPos42 = targPos12;
  targPos13 =  lU3;   targPos23 =  lD3;   targPos33 = targPos23 - 10;   targPos43 = targPos13;

  if (i < ird) {
    increment();
    i++;
  } else {
    setPrevPos();
    delay(phaseDelay);
    phase = 2;
    i = 0;
  }

  break;
case 2:
  // RR leg OUT       // FR leg OUT       // RL leg OUT            // FL leg OUT
  targPos11 =  lO1;   targPos21 =  lO1;   targPos31 = targPos21;   targPos41 = targPos11;
  targPos12 =  lO2;   targPos22 =  lO2;   targPos32 = targPos22;   targPos42 = targPos12;
  targPos13 =  lO3;   targPos23 =  lO3;   targPos33 = targPos23;   targPos43 = targPos13;

  if (i < ird) {
    increment();
    i++;
  } else {
    setPrevPos();
    delay(phaseDelay);
    phase = 3;
    i = 0;
  }

  break;
case 3:
  // RR leg DOWN      // FR leg UP        // RL leg UP           // FL leg DOWN
  targPos11 =  lD1;   targPos21 =  lU1;   targPos31 = targPos21;   targPos41 = targPos11;
  targPos12 =  lD2;   targPos22 =  lU2;   targPos32 = targPos22;   targPos42 = targPos12;
  targPos13 =  lD3;   targPos23 =  lU3;   targPos33 = targPos23;   targPos43 = targPos13;

  if (i < ird) {
    increment();
    i++;
  } else {
    setPrevPos();
    delay(phaseDelay);
    phase = 4;
    i = 0;
  }

  break;
case 4:
  // RR leg IN        // FR leg IN        // RL leg IN             // FL leg IN
  targPos11 =  lI1;   targPos21 = lI1;   targPos31 = targPos21;   targPos41 = targPos11;
  targPos12 =  lI2;   targPos22 = lI2;   targPos32 = targPos22;   targPos42 = targPos12;
  targPos13 =  lI3;   targPos23 = lI3;   targPos33 = targPos23;   targPos43 = targPos13;

  if (i < ird) {
    increment();
    i++;
  } else {
    setPrevPos();
    delay(phaseDelay);
    phase = 1;
    i = 0;
  }

  break;
```

```
  }

  updatePos();
}

// sitting angles
int fore1 = 0;    int fore2 = 20;   int fore3 = 25;
int hind1 = 0;    int hind2 = 45;   int hind3 = 30;

// sends PWM commands for dog to sit down
void sit() {
  Serial.println("Dog is sitting down.");
  moving = true;
  ird = 100;

  // RR leg                    // FR leg               // RL leg                   // FL leg
  targPos11 = hind1;        targPos21 = fore1;   targPos31 = hind1;       targPos41 = fore1;
  targPos12 = hind2;    targPos22 = fore2 - 7;   targPos32 = hind2 - 7;   targPos42 = fore2 + 5;
  targPos13 = hind3;    targPos23 = fore3 - 7;   targPos33 = hind3 - 7;   targPos43 = fore3 + 5;

  for (int s = 0; s <= ird; s++) {
    increment();
    updatePos();
  }
  delay(1000);
}

// runs between one function and another to make sure we start from "common" positions (i.e., initial positions)
// sends PWM commands to move joints into neutral standing position and then stop moving
void Stop() {
  Serial.println("Dog has stopped and is standing.");
  ird = 100;

  if (moving == true) {
    setPrevPos();
    // RR leg               // FR leg               // RL leg                    // FL leg
    targPos11 = initPos11;   targPos21 = initPos21;   targPos31 = initPos31;        targPos41 = initPos41;
    targPos12 = initPos12;   targPos22 = initPos22;   targPos32 = initPos32 - 15;   targPos42 = initPos42;
    targPos13 = initPos13;   targPos23 = initPos23;   targPos33 = initPos33 - 10;   targPos43 = initPos43;

    for (int s = 0; s <= ird; s++) {
      increment();
      updatePos();
    }

    moving = false; // once Stop() reaches its target positions
    delay(1000);
  } else if (moving == false) {
    setPrevPos();
    i = 0;     // reset increment counter for next function
    phase = 1; // reset phase indicator for next function
  }
}

// divides the difference between target positions and previous positions in small increments
void increment() {
  currPos11 += (targPos11 - prevPos11)/(ird);
  currPos12 += (targPos12 - prevPos12)/(ird);
  currPos13 += (targPos13 - prevPos13)/(ird);
  currPos21 += (targPos21 - prevPos21)/(ird);
  currPos22 += (targPos22 - prevPos22)/(ird);
  currPos23 += (targPos23 - prevPos23)/(ird);
  currPos31 += (targPos31 - prevPos31)/(ird);
  currPos32 += (targPos32 - prevPos32)/(ird);
  currPos33 += (targPos33 - prevPos33)/(ird);
```

```
  currPos41 += (targPos41 - prevPos41)/(ird);
  currPos42 += (targPos42 - prevPos42)/(ird);
  currPos43 += (targPos43 - prevPos43)/(ird);
}

// sets the current positions as previous positions for the next function
void setPrevPos() {
 prevPos11 = currPos11;
 prevPos12 = currPos12;
 prevPos13 = currPos13;
 prevPos21 = currPos21;
 prevPos22 = currPos22;
 prevPos23 = currPos23;
 prevPos31 = currPos31;
 prevPos32 = currPos32;
 prevPos33 = currPos33;
 prevPos41 = currPos41;
 prevPos42 = currPos42;
 prevPos43 = currPos43;
}

// moves to new calculated positions
void updatePos() {
  // rear right (4)
  setServo(12, currPos11); // RR hip roll
  setServo(13, currPos12); // RR hip pitch
  setServo(14, currPos13); // RR knee

  // front right (3)
  setServo(8, currPos21);  // FR hip roll
  setServo(9, currPos22);  // FR hip pitch
  setServo(10, currPos23); // FR hip knee

  // rear left (2)
  setServo(4, currPos31);  // RL hip roll
  setServo(5, currPos32);  // RL hip pitch
  setServo(6, currPos33);  // RL knee

  // front left (1)
  setServo(0, currPos41);  // FL hip roll
  setServo(1, currPos42);  // FL hip pitch
  setServo(2, currPos43);  // FL knee
}
```