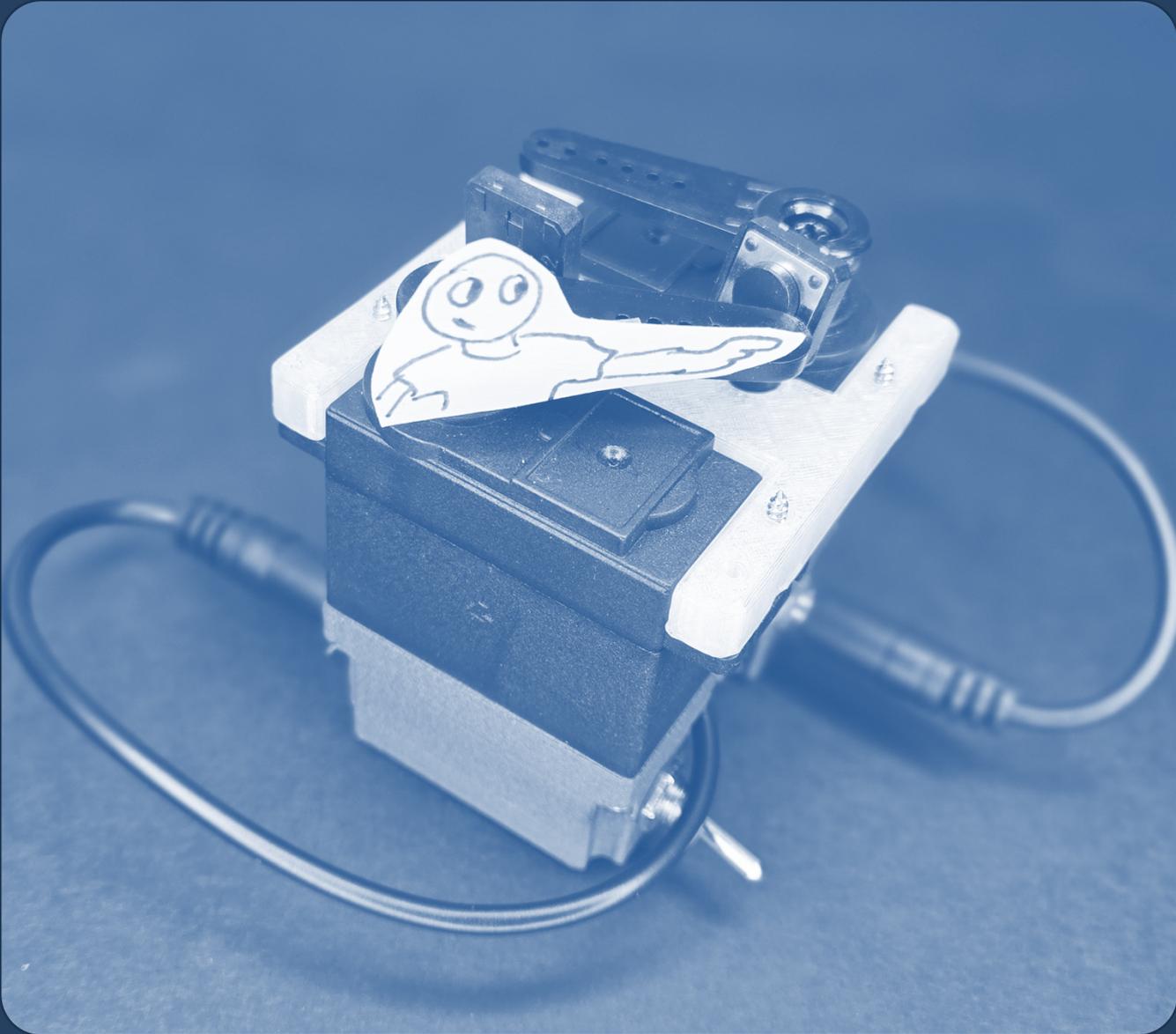


SMARTSERVO

DUAL SERVO: INTERACTIVE SHOW KIT



SMART SERVO PROJECT

DUAL SERVO: INTERACTIVE SHOW KIT

Version 1.0 | Published: June 17,2025 | Author: Judson Wagner, Wagner Labs LLC

CC Educational Use License

This guide is made available under a **Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0)** for educational purposes.

Creative Commons License:



You are free to:

- **Share** — copy and redistribute the material in any medium or format.
- **Adapt** — remix, transform, and build upon the material for educational purposes.

Under the following terms:

- **Attribution** - You must give appropriate credit to Wagner Labs LLC and the Smart Servo Project
- **NonCommercial** - You may not use this material for commercial purposes.
- **ShareAlike** - If you remix or adapt this material, you must distribute your contributions under the same license.
- **Educational Use Only** - This license is specifically limited to educational, academic, and non-profit educational institutions.

⚠ IMPORTANT DISTRIBUTION REQUIREMENTS

This cover page must be included with any distribution, reproduction, or posting of this guide. Any printed copies, electronic distributions, or online postings must include this complete cover page to maintain proper attribution and licensing terms.

Commercial Use & Smart Servo Requirement

For commercial use, bulk educational licensing, or use outside of traditional educational settings, please contact Wagner Labs LLC.

Hardware Requirement: This guide requires **Smart Servo devices** to complete the projects and activities described. Smart Servos are available through the Smart Servo Store and authorized distributors.

About the Smart Servo Project

The Smart Servo Project empowers inclusive innovation by providing accessible tools for creating assistive technologies and engaging STEM education. Our mission is to bridge technology and compassion through community-driven maker education.

Support our mission by purchasing Smart Servos and sharing our resources with your educational community.

Contact Information:

Judson Wagner | Wagner Labs LLC
Email: Judson@WagnerLabs.net
Website: WagnerLabs.net/SmartServo
Smart Servo Store: WagnerLabs..Store

Client: Emma Rodriguez, Age 9

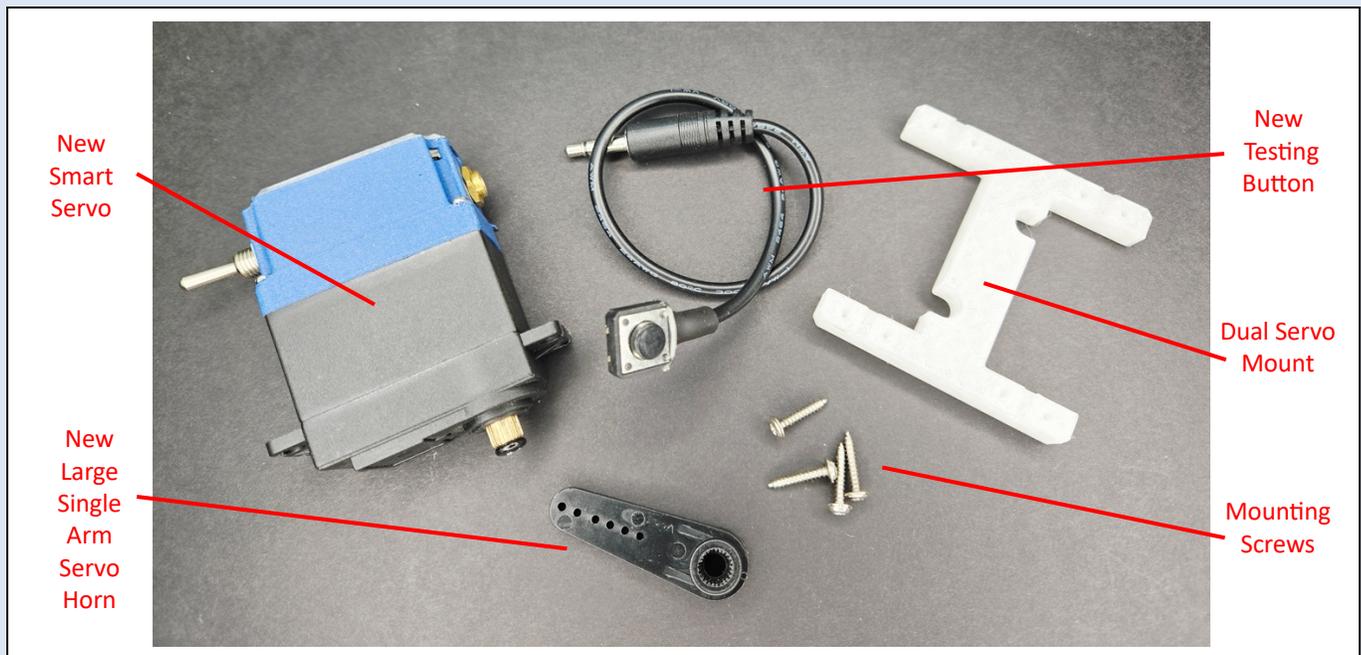
About Me: I'm a 4th grader who loves watching things move and interact. I have autism and sometimes find it hard to focus during quiet activities, but I'm fascinated by patterns and movements that seem alive and unpredictable.

My Challenge: I get restless during homework time and quiet periods at home. My parents are looking for something engaging that I can watch and interact with occasionally, but that doesn't require constant participation from me. I love the idea of things that seem to have personalities and move in interesting ways.

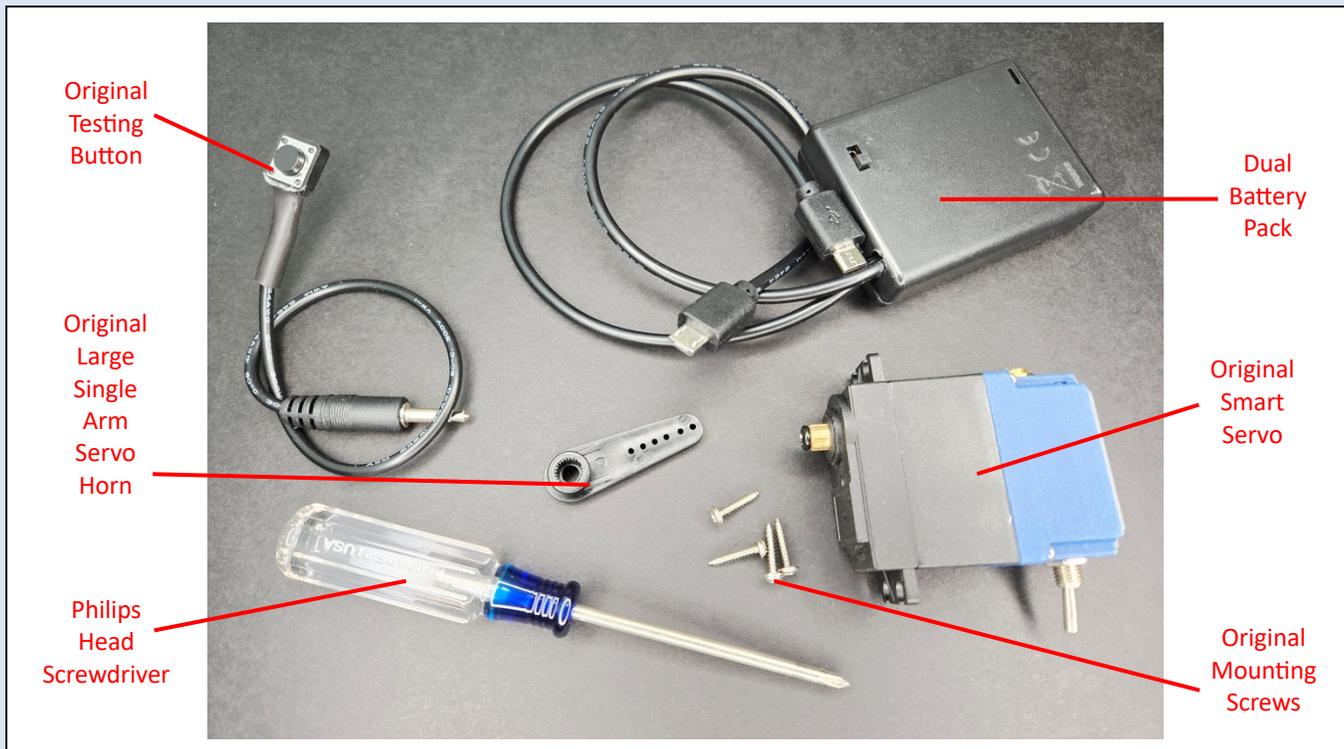
Technical Need: An entertaining dual-servo system that creates captivating, lifelike movements - one servo might slowly "sneak up" on the other, which then reacts with quick, startled movements. The unpredictable timing and human-like behaviors would provide engaging visual stimulation during quiet times.

Let's now investigate our kit and see if we can get started on something that can assist Emma.

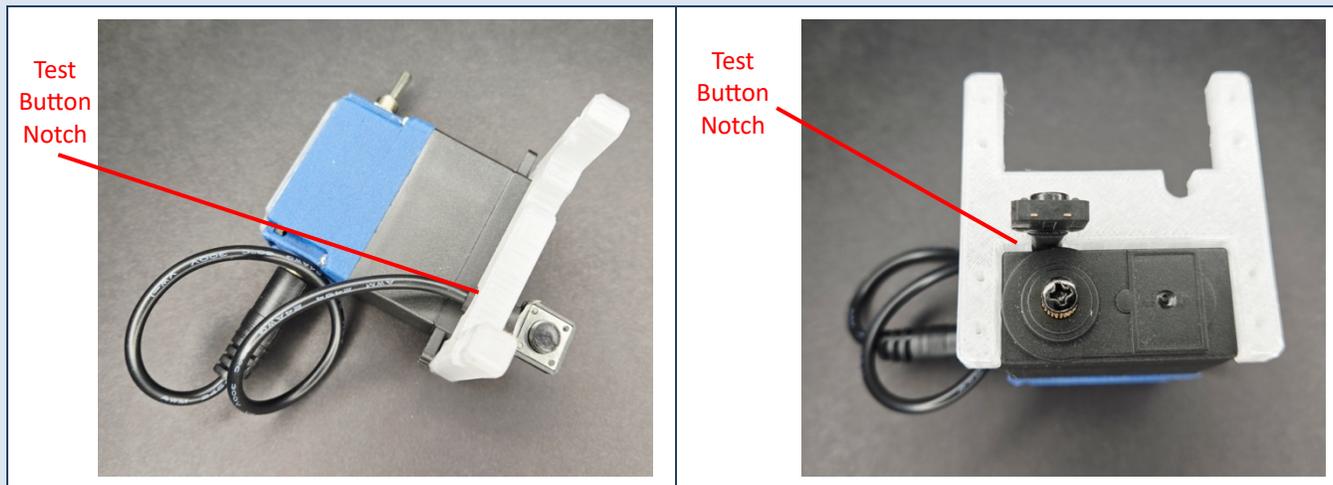
STEP 1: Lay out all the components that are new in this kit.



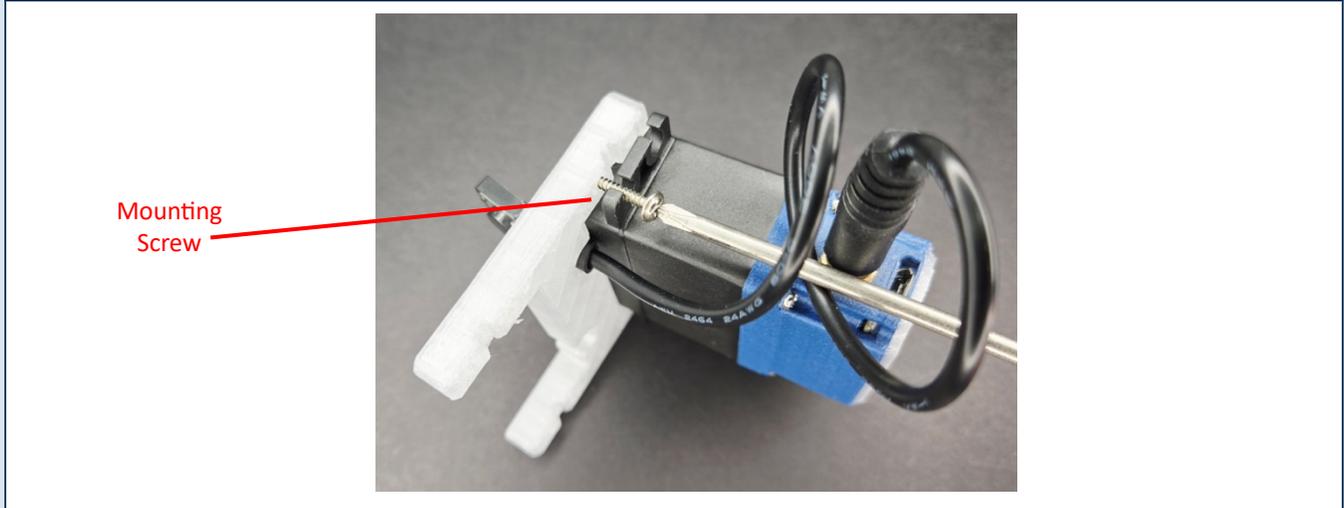
STEP 2: Make sure you have these items from your previous kits.



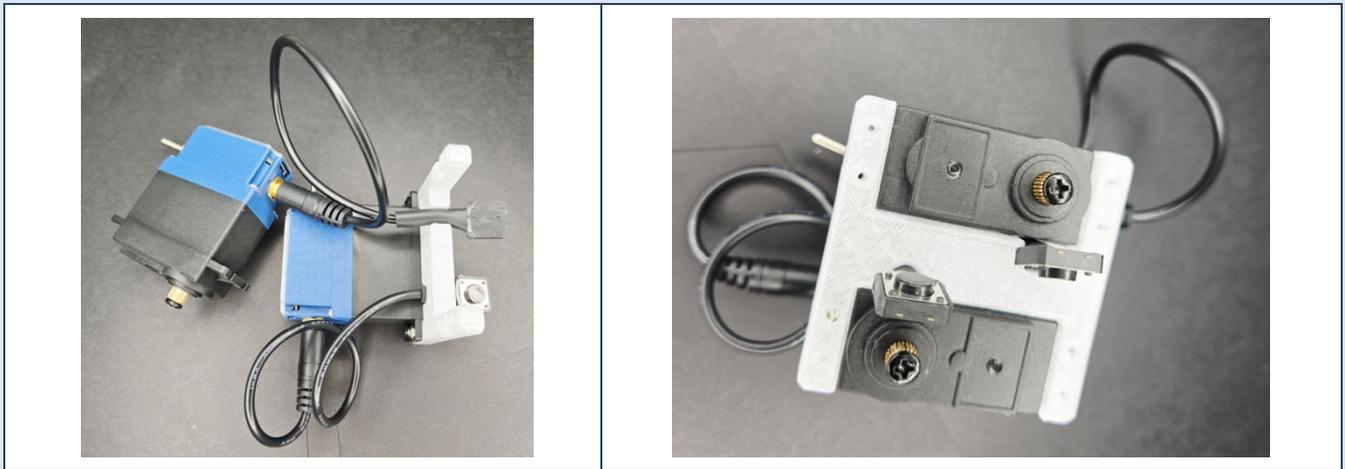
STEP 3: Connect a Testing Buttons to one of the Smart Servos and insert in into one of the Test Button Notches on the Dual Servo Mount. Then pin it in with the Smart Servo.



STEP 4: Secure the Smart Servo using the Mounting Screws.



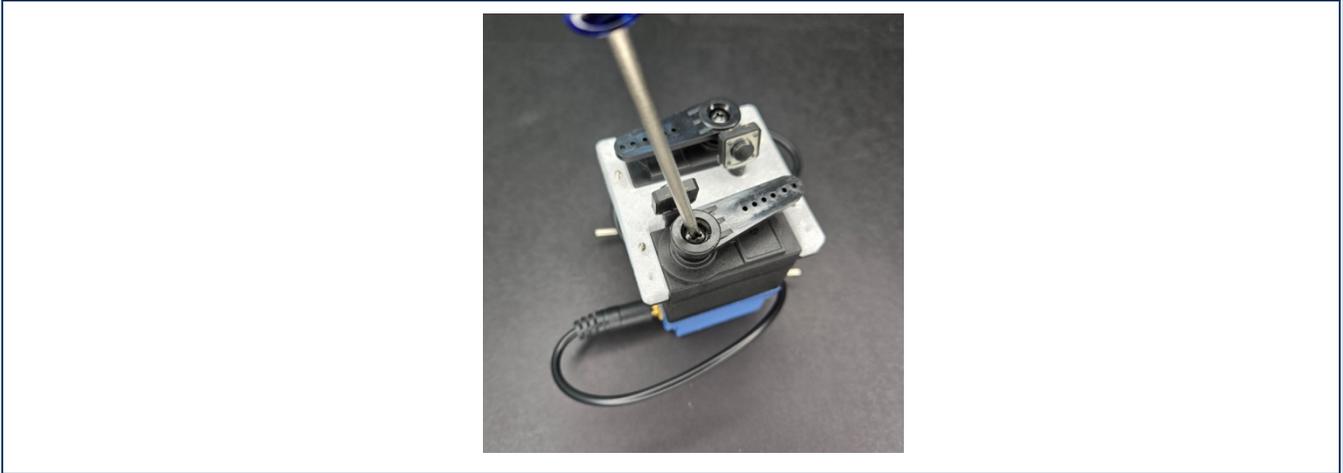
STEP 5: Repeat Steps 3 and 4 with the other Testing Button and Smart Servo.



STEP 6: Use the Mounting Screws to secure the second Smart Servo.



STEP 7: Connect the Large Single Arm Servo Horn to each of the Smart Servos.



STEP 8: Connect each Smart Servo to a computer and use the coding to determine the angle needed for the Horn from one Smart Servo to activate the button on the other Smart Servo. Record this angle for each Smart Servo.



STEP 9: Write a simple code (or modify Frank's Code) to see if you can create an ongoing repeating pattern where one button press initiates a first Smart Servo to move and press the button of the second Smart Servo which in turns activates it to push the button of the first servo. Tweak the program to optimize how long this back and forth button pressing can last.

STEP 10: Now we want to add some variability to this behavior and we're going to leverage AI (LLM) to do it. This is appropriate since LLMs, or Large Language Models, leverage randomness to imitate human intelligence through a process called sampling during text generation. This mechanism allows the model to select less probable, but more varied and creative, words rather than consistently choosing the single most likely word next.

The code at the end of this guide was generated by AI (Claude Sonnet 3.5) when give a working code like the one produced in Step 9 and the prompt below.



I have two Smart Servos connected to buttons, with each Smart Servo operating based on an attached button's state. When one button is pressed on the first Smart Servo, it triggers that Smart Servo to move and push the button attached to the second Smart Servo, creating a continuous back-and-forth chain reaction between the two servos. For reference, I'm attaching some Circuit Python code (code.py) that I've been using with these Smart Servos.

I need Circuit Python code where the servo movements have personality - like two young siblings playfully and literally 'pushing each other's buttons.' I want their behavior to be random and varied so that sometimes they're sneaky, sometimes they're reactive and swift, sometimes they're patient and biding their time, and sometimes they tease and fake before finally pushing. The goal is to keep this playful back-and-forth going indefinitely, with each servo's movement feeling organic and unpredictable rather than mechanical.

What we did here is called Vibe Coding. Check out the attached code and/or see if a similar prompt will give you an interesting behavior interaction between your Smart Servos.

CONGRATULATIONS!

You just coordinated two servos to create lifelike interactive motion. Next, we'll want to consider what needs to be done next to better assist Emma. Read her profile again and think about what captures her attention. What follow-up questions would you ask her family about the kinds of movements and patterns she finds most engaging? How might you adjust the timing or behaviors to match her interests?



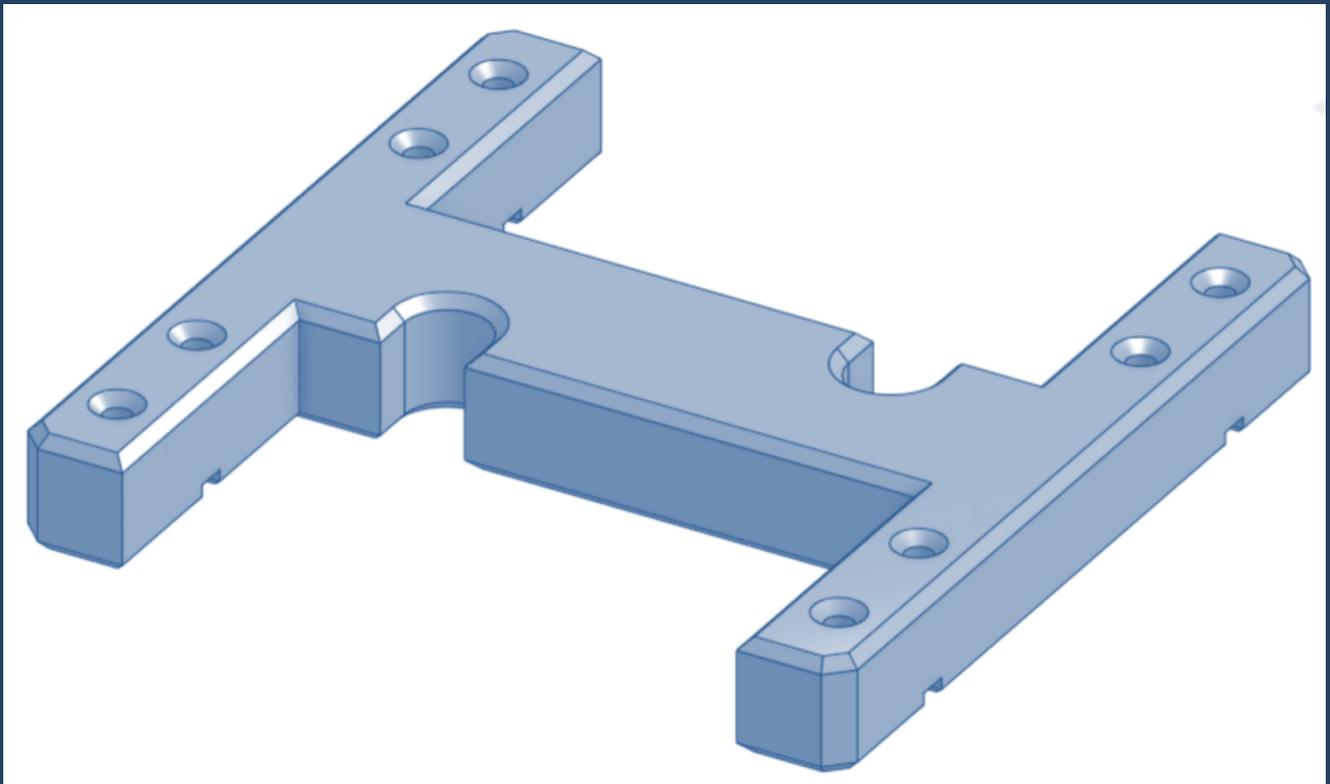
REMINDER ABOUT CODING SNIPS

If you want to return your code to the original “factory setting”, just copy and paste from here: tinyurl.com/SmartServoSnips



3D PRINTING FILES

If you're able to 3D Print, download the 3D parts used in this project here: tinyurl.com/SS-STL-DUAL





THE BIGGER PICTURE

CREATING LIFELIKE MOTION IN YOUR INTERACTIVE SHOW

What Makes Something Seem Alive?

When your two servos play "tag," something remarkable happens - simple motors following basic code suddenly seem to have personalities. One "sneaks up" slowly and cautiously. The other "startles" and jerks away quickly. Even though you know they're just machines, your brain can't help seeing intention and life. This isn't a bug in perception - it's how we understand the world, and engineers exploit it to create compelling experiences.

The secret lies in motion patterns. When something moves with hesitation then sudden bursts, we read it as cautious or fearful. Smooth, deliberate motion suggests confidence. When something pauses then reacts to another object's movement, we see awareness and social interaction. Your tag-playing servos tap into these deep-rooted perceptual systems.

The Principles of Character Animation

Disney animators discovered these principles decades ago. They found realistic motion looked dead and boring - characters needed exaggeration, anticipation, and follow-through to seem alive. Before a character jumps, they crouch down ("anticipation"). When they land, parts continue moving ("follow through"). Even simple movements need "ease in" and "ease out" - starting slowly, speeding up, then slowing down.

Your tag system uses these exact principles. The "sneaking" servo moves slowly with pauses. The "startled" servo accelerates quickly then decelerates. These patterns mimic how real living things move because of physics (mass and inertia) and biology (muscles that gradually contract and relax).

Social Robotics and Perceived Personality

Engineers designing social robots obsess over these motion qualities. Robot vacuums that move dejectedly when their bin is full aren't really sad, but the motion pattern makes you feel sympathetic. Research shows people respond dramatically differently to robots based purely on movement personality. Fast, jerky motions seem aggressive. Slow, smooth motions seem calm. Asymmetric, slightly irregular movements seem more natural - real living things are never perfectly regular.

Engineering Personality Into Motion

Creating these effects requires thinking beyond mechanics into psychology. You're not just programming positions and speeds - you're choreographing a performance. Should the "scared" servo pause before reacting, or respond instantly? Should the "sneaky" servo move in bursts with pauses, or continuously? Each choice shapes personality and emotional impact.

This intersection of engineering and character design appears in video games, animated films, theme parks, and consumer robotics. Emma's interactive show demonstrates these principles perfectly - unpredictable timing keeps it engaging, personality-filled movements create emotional connection, and the visual focus provides sensory input without requiring constant participation.

AI Generated Circuit Python (Claude Sonnet 3.5)

```
import time
import board
from digitalio import DigitalInOut, Direction, Pull
import pwmio
import servo
import random

import adafruit_dotstar
pix = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)

button = DigitalInOut(board.D2)
button.direction = Direction.INPUT
button.pull = Pull.UP

switch = DigitalInOut(board.D1)
switch.direction = Direction.INPUT
switch.pull = Pull.DOWN

pwm = pwmio.PWMOut(board.A2, duty_cycle=2 ** 15, frequency=50)
servo1 = servo.Servo(pwm)

def random_servo_movement(start_angle, end_angle, max_time):
    # Initialize variables
    start_time = time.monotonic()
    current_angle = start_angle

    # Generate intermediate waypoints between start and end
    num_waypoints = random.randint(3, 8)

    # Create a list of possible positions between start and end
    min_angle = min(start_angle, end_angle)
    max_angle = max(start_angle, end_angle)
    angle_range = max_angle - min_angle

    # Generate random waypoints and sort them appropriately
    waypoints = [min_angle + random.random() * angle_range for _ in
range(num_waypoints)]

    # Add start and end points, and sort accordingly
    if start_angle < end_angle:
        waypoints = [start_angle] + sorted(waypoints) + [end_angle]
    else:
        waypoints = [start_angle] + sorted(waypoints, reverse=True) +
[end_angle]

    # Remove any duplicates
```

*COPY & PASTE
this code from:
tinyurl.com/ss-code-dual*

```

waypoints = [waypoints[i] for i in range(len(waypoints))
              if i == 0 or abs(waypoints[i] - waypoints[i-1]) > 1]

# Ensure the last waypoint is exactly the end_angle
waypoints[-1] = end_angle

# Allocate random time for each segment
segment_times = []
for _ in range(len(waypoints) - 1):
    # Add more variation to timing
    segment_times.append(random.uniform(0.2, 1.5))

# Scale time to fit within max_time
total_time = sum(segment_times)
segment_times = [t * (max_time / total_time) for t in segment_times]

# Execute the movement plan
for i in range(len(waypoints) - 1):
    start_pos = waypoints[i]
    end_pos = waypoints[i+1]
    duration = segment_times[i]

    # Occasional pause or hesitation
    is_pause = random.random() < 0.2

    # Execute segment
    segment_start = time.monotonic()
    while True:
        elapsed = time.monotonic() - segment_start
        if elapsed >= duration:
            break

        if not is_pause:
            # Calculate position based on elapsed time
            progress = elapsed / duration
            current_angle = start_pos + progress * (end_pos -
start_pos)
            servo1.angle = current_angle

        # Small delay to prevent CPU hogging
        time.sleep(0.01)

    # Explicitly set to end position of this segment
    servo1.angle = end_pos

# Double-check final position explicitly
servo1.angle = end_angle

```

```
# Return the total time taken
return time.monotonic() - start_time

state = 0
while True:
    pix[0] = (0,100,0)
    if switch.value == 1 and button.value == 0 and state == 0:
        # Display blue light while servo is moving to 180
        pix[0] = (0,0,100)

        # Occasional 'teasing' movement before main move
        if random.random() < 0.3:
            random_servo_movement(0, 90, 1.0)
            time.sleep(0.5)

        # Move to 180 degrees with random speed variations and possible
        # pauses
        random_servo_movement(0, 180, 5.0) # Start at 0, end at 180, max
        # 5 seconds

        # Set final position one more time to ensure we're at 180
        servo1.angle = 180
        time.sleep(0.1) # Small delay to ensure servo reaches position

        # Immediately return to 0 degrees (as fast as possible)
        servo1.angle = 0
        state = 1
        pix[0] = (0,100,0)
    elif switch.value == 1 and button.value == 1 and state == 1:
        servo1.angle = 0
        pix[0] = (0,100,0)
        state = 0
        time.sleep(0.0)
    elif switch.value == 0 and button.value == 0:
        servo1.angle = 0
        pix[0] = (125, 115, 3)
        time.sleep(0.5)
    elif switch.value == 0 and button.value == 1:
        servo1.angle = 0
        for i in range (10,255,1):
            pix[0] = (i,0,0)
        for i in range (255,10,-1):
            pix[0] = (i,0,0)
        time.sleep(1.0)
    time.sleep(0.0)
```

