ES50 Project:

# EMG-Triggered Robot with Integrated Sensor Feedback- Gigachad Bro-bot by Flexecutioners

## Abstract:

\The small step of automating a Boe Bot opens up the possibility of using bio-sensing for automation of significantly more complex systems.

## Introduction:

### I. System Overview and Core Objective

This project realizes a Human-Machine Interface (HMI) enabling control of a Boe-Bot mobile robot through non-invasive biological signals, specifically Electromyography (EMG) which is derived from voluntary muscle contractions. The fundamental objective is to translate the complex, analog nature of EMG signals into discrete, reliable commands for robotic locomotion (single steps), while critically incorporating environmental feedback via ultrasonic sensors to ensure safe operation. The system architecture integrates signal acquisition hardware (SpikerBox), real-time digital signal processing (Python script), and embedded control (Arduino MKR Zero) managing actuators and visual feedback elements (LEDs). This project utilizes a complete cycle- biological signal sensing to controlled physical action, actualizing core principles of electrical engineering and digital systems.

EMG (Electromyography) measures the electrical activity produced by muscles when they contract. When a muscle fiber is activated—either by voluntary movement or reflex—a small electrical signal is generated. EMG sensors (usually surface electrodes) pick up these signals through the skin. The stronger the muscle contraction, the higher the amplitude of the signal. These signals are then amplified and processed to analyze muscle activity or trigger external devices like robots. We then use this amplified signals, process them using methods such as band limitation and filtering to obtain a signal tha is able to give up important information on the current biological state of the human.

Along with the movement of the boe bot, an additional LED feature was added based off of the signals. At high frequencies there were green lights that would be flashed, these flashes would typically be paired with the movement of the Boe Bot. However, at low frequencies, a red light is flashed instead.

## Design:

Backyardbrains Spikerbox

**II. Analog Signal Acquisition and Conditioning (SpikerBox Subsystem)**

The initial stage involves capturing the faint electrical potentials generated by muscle fiber activity and preparing them for digital conversion.

- A. EMG Sensing and Transduction

The process originates with surface electrodes placed on the user's skin (e.g., forearm) that detect the voltage differentials produced during muscle contraction. These raw signals are typically in the microvolt to low millivolt range and are superimposed with noise from various sources (biological and environmental).

- B. Analog Front-End Processing (Within SpikerBox)

The captured raw EMG signal is unsuitable for direct digital processing and requires significant analog conditioning, as evidenced by the SpikerBox schematic. This conditioning typically involves several operational amplifier (Op-Amp) based stages:

➔ Instrumentation Amplification: Bio-potentials often require differential amplification to reject common-mode noise (noise appearing identically on both

sensing electrodes). An instrumentation amplifier (potentially the AD8293 listed) provides high gain and high common-mode rejection ratio (CMRR).

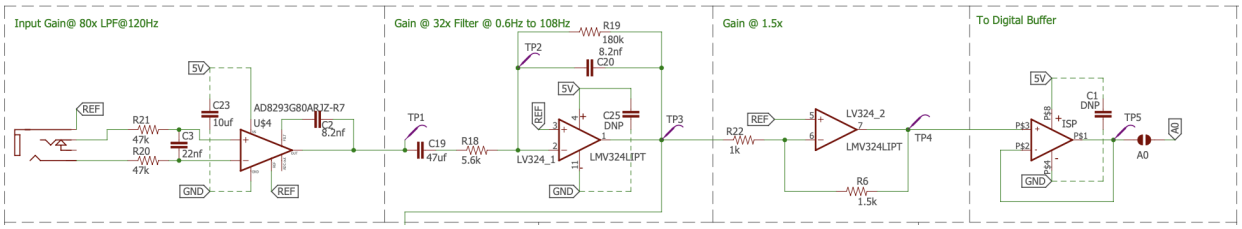
➔ Filtering (Low-Pass): High-frequency noise (above the physiological range of EMG, typically < 500 Hz) is removed. The schematic indicates a Low-Pass Filter (LPF) stage with a cutoff around 120Hz ("LPF@120Hz") likely implemented as an active filter using Op-Amps (like the LMV324) in conjunction with resistors (R) and capacitors (C).

The impedance of a capacitor, ZC = 1 / (jωC), where ω = 2πf, becomes very small at high frequencies, effectively shorting high-frequency noise to ground or a reference in typical RC filter configurations (as discussed conceptually for passive RC filters). Active filters allow for gain and sharper cutoff characteristics.

➔ Filtering (Notch): Power line interference (typically 60Hz in the US or 50Hz elsewhere) is a significant noise source. A dedicated Notch Filter ("Notch Filter@60Hz") specifically targets and attenuates this frequency. This is often implemented using specialized active filter topologies (e.g., Twin-T or state-variable filters) built around Op-Amps. Filtering (Band-Pass): The signal is further refined by a Band-Pass Filter (BPF) stage ("Filter @ 0.6Hz to 108Hz"). This stage removes very low-frequency baseline drift (below 0.6Hz) and further attenuates high frequencies (above 108Hz), isolating the most relevant EMG frequency components. This is again achieved using active filter design principles.

➔ Gain Stages: Multiple stages provide amplification ("Gain @ 80x", "Gain @ 32x", "Gain @ 1.5x") to bring the millivolt-level signal up to a voltage range suitable for the ADC (e.g., 0-5V or 0-3.3V).

➔ Buffering: A final Op-Amp Buffer (Voltage Follower) stage ("To Digital Buffer") ensures that the ADC input does not significantly load the preceding filter stages, preserving the signal integrity. A buffer provides a very high input impedance and a low output impedance.




- C. Analog-to-Digital Conversion (ADC)

The conditioned analog EMG voltage is converted into a digital representation by the ADC integrated within the SpikerBox's microcontroller (ATmega328, according to the Heart & Brain SpikerBox firmware). Sampling: The process of measuring the analog voltage at discrete, regularly spaced points in time.

The sampling rate (fs) is determined by the Arduino Timer1 interrupt configuration. The firmware sets interrupt_Number=199, which corresponds to a sampling rate of **fs = 16×10^6 Hz / ((199+1)×8) = 10,000 Hz**. According to the Nyquist-Shannon Sampling Theorem, to accurately represent a signal without aliasing, the sampling rate fs must be strictly greater than twice the maximum frequency component (fmax) present in the signal **(fs > 2fmax)**.

The analog filtering stages (max ~108Hz) ensure this condition is met, preventing higher frequencies from folding back and distorting the representation of the desired EMG band.

> **Quantization:** The conversion of the continuous sample voltage into a discrete digital value. The ATmega328 ADC has a 10-bit resolution, meaning it divides the input voltage range (typically 0V to the reference voltage, AREF, often 5V or internal references) into 2^10 = 1024 distinct levels. Each sample is assigned the binary code corresponding to the closest level. This process introduces quantization error, which is the difference between the actual analog voltage and the voltage represented by the chosen digital level. The smallest voltage change resolvable is the step size (Δ), approximately Δ = VoltageRange / (2^b - 1) or Δ = VoltageRange / 2^b, where b=10.

**III. Data Encoding and Serial Transmission (SpikerBox -> Python)**

The 10-bit digital samples from the ADC need to be transmitted over a standard serial interface, which typically handles 8-bit bytes.

- A. Binary Representation and Bitwise Operations-

The 10-bit ADC result is a binary integer. To transmit it, the Arduino firmware employs bitwise operations: tempSample >> 7:

Right-shifts the 10-bit value by 7 positions, isolating the 3 Most Significant Bits (MSBs). | 0x80: Performs a bitwise OR with 10000000 binary.

This sets the MSB of the first byte to 1, marking it as the start of a 2-byte sample frame. tempSample & 0x7F: Performs a bitwise AND with 01111111 binary. This masks out (zeros) the MSB and keeps the 7 Least Significant Bits (LSBs) for the second byte.

This custom encoding scheme packs the 10 bits of data into two 7-bit payloads within two 8-bit bytes, using the MSB of the first byte for synchronization.

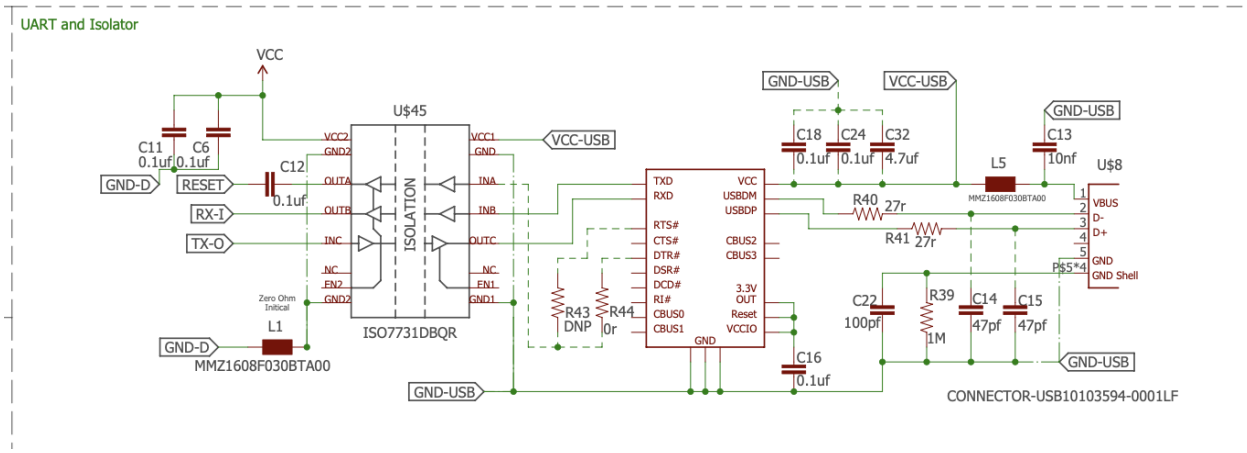
- B. UART Communication

The encoded bytes are sent serially to the Python host using the Universal Asynchronous Receiver/Transmitter (UART) protocol principles, although implemented over USB CDC (Communication Device Class) which emulates a serial port.

Asynchronous: No separate clock signal is transmitted alongside the

data. Both SpikerBox (firmware) and Python (pyserial) must be configured to the same Baud Rate (222222 bits per second in this project's configuration) to correctly time the sampling of bits.

Data Framing: While standard UART uses start/stop bits, this project uses the MSB marker bit in the custom encoding for frame synchronization. The Python receiver must continuously read bytes and identify the marker bit to know where a 10-bit sample begins across two bytes.




**IV. Real-Time Signal Processing and Control (Python)**

The Python script receives the raw byte stream and processes it to detect muscle flexions reliably.

- A. Data Parsing and Reconstruction

The parse_and_process_data function implements the reverse of the encoding process: It reads bytes from the buffer managed by the read_thread. It identifies the start byte (MSB=1) and the following data byte (MSB=0).

It reconstructs the 10-bit value using bitwise AND (&), Left Shift (<<), and OR (|) operations:

raw_value = ((byte1 & 0x7F) << 7) | (byte2 & 0x7F).

It applies an offset (sample_value = raw_value - 512) to center the data, converting the unsigned ADC range (0-1023) into a signed-like representation (-512 to +511) useful for thresholding fluctuations around a baseline.

- B. Digital Filtering

Hysteresis and Debouncing Raw EMG data is noisy. Simple thresholding can lead to spurious triggers. The script implements two techniques:

➔ Hysteresis (Combinational Logic):

Uses two thresholds (**FLEX_THRESHOLD_HIGH, FLEX_THRESHOLD_LOW**). A "flex" is only initiated when the signal exceeds the high threshold, and the system only resets (becomes ready for a new flex) after the signal drops below the low threshold. This prevents oscillations if the signal hovers near a single threshold.

The core operation involves magnitude comparators (implemented in software as sample_value > HIGH_THRESH, sample_value < LOW_THRESH).

➔ Debouncing (Sequential Logic):

Uses collections.deque (fixed-size queues) as rudimentary shift registers to store the boolean results of the threshold comparisons over the last DEBOUNCE_VALUES samples. A flex state is confirmed only if all() samples in flex_confirm_queue are true (signal consistently above high threshold).

This is a temporal AND operation. A rest state is confirmed only if all() samples in rest_confirm_queue are true (signal consistently below low threshold). This dependence on a sequence of past comparison results makes it a form of sequential logic, filtering out brief noise spikes that don't persist for DEBOUNCE_VALUES samples.

- C. State Management (Finite State Machine)

The control logic uses the can_trigger_step boolean variable to implement a **2-state Finite State Machine** (FSM), ensuring only **one step command** is sent per valid flex-then-relax cycle:

**State**: **READY** (can_trigger_step = True): Waiting for a confirmed flex (hysteresis + debounce condition met). **Input**: Confirmed Flex. **Transition**: READY -> WAITING_RESET.

**Action**: Send 'P' command, set can_trigger_step = False.

**State: WAITING_RESET** (can_trigger_step = False): Waiting for a confirmed rest (signal below low threshold for debounce duration). **Input**: Confirmed Rest. **Transition**: WAITING_RESET ->READY. **Action**: Set can_trigger_step = True.

This FSM utilizes the **can_trigger_step variable** as its state memory, conceptually equivalent to a **single D flip-flop** storing the system's readiness state.

```
import threading
import serial
import time
import matplotlib.pyplot as plt
```

```
import matplotlib.animation as animation
import numpy as np
import collections

# Configuration
# SpikerBox Connection
SPIKERBOX_PORT = '/dev/cu.usbserial-DM8UFWOE' # SpikerBox port
SPIKERBOX_BAUD = 222222                       #  Baud Rate

# Boe-Bot Arduino Connection (Receives 'P' command)
CAR_ARDUINO_PORT = '/dev/cu.usbmodem1101' # Boe-Bot's port (MKR Zero)
CAR_ARDUINO_BAUD = 9600

# Processing & Control (Hysteresis for Step Triggering)
FLEX_THRESHOLD_HIGH = 400
FLEX_THRESHOLD_LOW = 100  # RESET trigger
DEBOUNCE_VALUES = 2
# Plotting
PLOT_HISTORY_LENGTH = 3000 # Number of samples to display (approx 0.3s)
PLOT_UPDATE_INTERVAL_MS = 50 # How often to redraw plot (milliseconds)
SAMPLE_RATE_HZ = 10000

# Global Variables
connected = False
spikerbox_serial = None
arduino_for_car = None # For Boe-Bot
input_buffer = bytearray()
buffer_lock = threading.Lock()
sample_history = collections.deque(maxlen=PLOT_HISTORY_LENGTH)
stop_thread = False

#  State Variables
can_trigger_step = True # Flag to allow triggering a new step
flex_confirm_queue = collections.deque(maxlen=DEBOUNCE_VALUES)
rest_confirm_queue = collections.deque(maxlen=DEBOUNCE_VALUES)

#  Serial Port Reading Thread
def read_from_port():
    global connected, input_buffer, spikerbox_serial, stop_thread
    print(f"Thread: Attempting to connect to {SPIKERBOX_PORT} at
{SPIKERBOX_BAUD} baud...")
    try:
        spikerbox_serial = serial.Serial(SPIKERBOX_PORT, SPIKERBOX_BAUD,
```

```
timeout=0.1)
        print("Thread: SpikerBox Connected.")
        connected = True
        spikerbox_serial.reset_input_buffer()
    except serial.SerialException as e:
        print(f"Thread: ERROR connecting to SpikerBox - {e}")
        connected = False
        return

    while not stop_thread:
        try:
            if spikerbox_serial.is_open and spikerbox_serial.in_waiting >
0:
                reading =
spikerbox_serial.read(spikerbox_serial.in_waiting)
                if reading:
                    with buffer_lock:
                        input_buffer.extend(reading)
            elif not spikerbox_serial.is_open:
                 print("Thread: SpikerBox Serial port closed.")
                 break
            time.sleep(0.005)
        except serial.SerialException as e:
            print(f"Thread: SpikerBox Serial Error - {e}")
            break
        except Exception as e:
            print(f"Thread: SpikerBox Unexpected Error - {e}")
            time.sleep(0.1)

    if spikerbox_serial and spikerbox_serial.is_open:
        spikerbox_serial.close()
    print("Thread: Read thread finished.")

#  Data Parsing and Processing
def parse_and_process_data():
    """Parses data, handles step triggering, sends command to Boe-Bot."""
    global input_buffer, sample_history, can_trigger_step

    processed_bytes = 0
    new_samples = []

    with buffer_lock:
        buf_len = len(input_buffer)
```

```
        i = 0
        while i < (buf_len - 1):
            byte1 = input_buffer[i]
            byte2 = input_buffer[i+1]

            if (byte1 & 0x80) == 0x80:
                if (byte2 & 0x80) == 0x00:
                    msb_part = (byte1 & 0x7F) << 7
                    lsb_part = byte2 & 0x7F
                    raw_value = msb_part | lsb_part
                    sample_value = raw_value - 512
                    new_samples.append(sample_value)

                    #  START: STEP TRIGGER LOGIC
                    is_potentially_flexing = sample_value >
FLEX_THRESHOLD_HIGH
                    is_potentially_resting = sample_value <
FLEX_THRESHOLD_LOW

                    flex_confirm_queue.append(is_potentially_flexing)
                    rest_confirm_queue.append(is_potentially_resting)

                    is_flex_confirmed = all(flex_confirm_queue)
                    is_rest_confirmed = all(rest_confirm_queue)

                    command_to_send_bb = None # Command for Boe-Bot

                    #  Condition to Trigger ONE Step
                    if can_trigger_step and is_flex_confirmed:
                        print(f"*** Flex Trigger Detected (Val:
{sample_value} > {FLEX_THRESHOLD_HIGH}) ***", flush=True)
                        command_to_send_bb = b'P' # Send 'P' for Pulse/Step
                        can_trigger_step = False # Prevent immediate
re-triggering
                        print(">>> Sending Step Command 'P' to Boe-Bot
<<<", flush=True)

                    #  Condition to Re-enable Triggering
                    elif not can_trigger_step and is_rest_confirmed:
                         can_trigger_step = True # Re-enable triggering
                         for _ in range(DEBOUNCE_VALUES):
                             flex_confirm_queue.append(False)
                             rest_confirm_queue.append(True)
```

```
                    #  Send Command to Boe-Bot Arduino
                    if command_to_send_bb and arduino_for_car and
arduino_for_car.is_open:
                        try:
                            arduino_for_car.write(command_to_send_bb)
                        except serial.SerialException:
                            print("Error writing Step command to Boe-Bot.",
flush=True)
                    #  END: STEP TRIGGER LOGIC

                    processed_bytes = i + 2
                    i += 2
                else: # Invalid second byte
                    processed_bytes = i + 1
                    i += 1
            else: # Not a start byte
                processed_bytes = i + 1
                i += 1

        # Remove processed bytes
        if processed_bytes > 0:
             input_buffer = input_buffer[processed_bytes:]

    # Add newly parsed samples to history
    if new_samples:
        sample_history.extend(new_samples)


#  Plotting
fig, ax = plt.subplots()
plot_x_axis = np.linspace(-(PLOT_HISTORY_LENGTH / SAMPLE_RATE_HZ), 0,
num=PLOT_HISTORY_LENGTH)
line, = ax.plot(plot_x_axis, np.zeros(PLOT_HISTORY_LENGTH), 'b-',
label='EMG Signal')
threshold_high_line = ax.axhline(FLEX_THRESHOLD_HIGH, color='r',
linestyle='--', label=f'Step Thr ({FLEX_THRESHOLD_HIGH})')
threshold_low_line = ax.axhline(FLEX_THRESHOLD_LOW, color='g',
linestyle=':', label=f'Reset Thr ({FLEX_THRESHOLD_LOW})')
ax.set_ylim(-550, 550)
ax.set_xlim(-(PLOT_HISTORY_LENGTH / SAMPLE_RATE_HZ), 0)
ax.set_xlabel(f"Time (seconds)")
ax.set_ylabel("EMG Value (approx)")
```

```
ax.set_title("EMG Step Control (No Sensors)") # Updated title
ax.legend(loc='upper left')
fig.tight_layout()

# Animation update function
def update_plot(frame):
    parse_and_process_data() # Parse data, trigger steps
    plot_data = list(sample_history)
    if len(plot_data) < PLOT_HISTORY_LENGTH:
        padding = [0] * (PLOT_HISTORY_LENGTH - len(plot_data))
        plot_data = padding + plot_data
    line.set_ydata(plot_data)
    return line, threshold_high_line, threshold_low_line

#  Main Execution
if __name__ == "__main__":
    # Initialize queues
    for _ in range(DEBOUNCE_VALUES):
        flex_confirm_queue.append(False)
        rest_confirm_queue.append(True)
    for _ in range(PLOT_HISTORY_LENGTH):
         sample_history.append(0)

    #  Connect to Boe-Bot Arduino
    try:
        arduino_for_car = serial.Serial(CAR_ARDUINO_PORT, CAR_ARDUINO_BAUD,
timeout=1)
        time.sleep(2)
        print(f"Connected to Boe-Bot Arduino on {CAR_ARDUINO_PORT}")
    except serial.SerialException as e:
        print(f"Warning: Could not open Boe-Bot Arduino port
{CAR_ARDUINO_PORT}. {e}")
        arduino_for_car = None

    # Start the SpikerBox reading thread
    read_thread = threading.Thread(target=read_from_port, daemon=True)
    read_thread.start()

    time.sleep(1.5) # Give thread time to connect

    if not connected:
        print("Failed to connect to SpikerBox. Exiting.")
    elif not arduino_for_car:
```

```
         print("Failed to connect to Boe-Bot Arduino. Running without
Boe-Bot control.")
    # Only proceed with plotting if SpikerBox connected
    if connected:
        try:
            print("Starting plot animation...")
            ani = animation.FuncAnimation(fig, update_plot,
interval=PLOT_UPDATE_INTERVAL_MS,
                                          blit=True,
cache_frame_data=False, save_count=50)
            plt.show() # Blocks here until plot window is closed

        except Exception as e:
            print(f"Error during plotting: {e}")
        finally:
            print("Plot window closed or error occurred. Stopping...")
            stop_thread = True
            if read_thread.is_alive():
                read_thread.join(timeout=2)
            if read_thread.is_alive():
                 print("Warning: Read thread did not terminate cleanly.")

            # Final cleanup - Close Boe-Bot port if open
            if arduino_for_car and arduino_for_car.is_open:
                try:
                    pass # No final command needed for step mode
                except: pass
                finally:
                    arduino_for_car.close()
                    print("Boe-Bot Arduino connection closed.")

            plt.close(fig)
            print("Script finished.")
    else:
         # If SpikerBox connection failed initially
         if read_thread.is_alive():
              stop_thread = True
              read_thread.join(timeout=1)
         print("Exiting due to SpikerBox connection failure.")
```

**V. Command Transmission and Robot Control (Python → Arduino → Boe-Bot)**

- A. Serial Communication (UART)

The Python script sends the single-byte command b'P' to the Arduino MKR Zero via a second UART link (Port CAR_ARDUINO_PORT, Baud Rate 9600) using pyserial.

- B. Arduino Control Logic (Embedded FSM)

The Arduino code running on the MKR Zero acts as the real-time robot controller, implicitly implementing its own FSM: Inputs: Receives serial command ('P') from Python, reads distance values from ultrasonic sensors (using pulseIn, which measures time based on the microcontroller's clock frequency).

States (Implicit): **IDLE/STOPPED** (Red LEDs), **MOVING** (Green LEDs), **potentially BLOCKED** (Red LEDs, no movement).

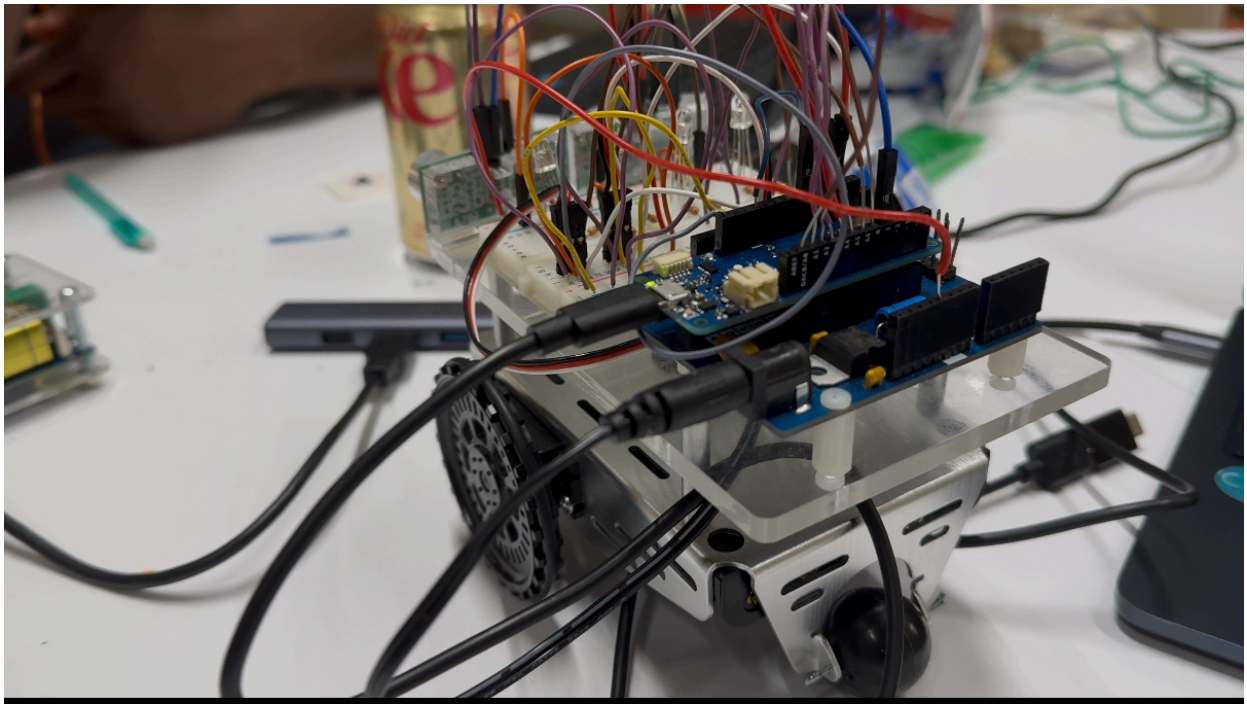
**Combinational Logic:** Compares sensor distance values against a safety threshold (distance > 15cm). This is the integrated Sensor logic for the gigachad Boe bot

**Sequential Logic/State Transitions:** In IDLE state, upon receiving 'P': Checks sensor readings.

**If clear:** Transitions to MOVING state. If blocked: Stays in IDLE (or enters BLOCKED state).

**In MOVING state**: Executes fixed-duration motor movement. After STEP_DURATION_MS (timed using delay()), transitions back to IDLE state.

(Boe bot won’t move unless path is clear)



**Outputs: Actuator Control (Servos):** Uses the Servo library's writeMicroseconds() function.

This generates Pulse Width Modulated (PWM) signals. For continuous rotation servos, the pulse width determines speed/direction relative to a calibrated stop value (LEFT_STOP_US, RIGHT_STOP_US).

This is a form of digital-to-analog control for the motor speed.

**Visual Feedback (LEDs):** Uses digitalWrite() to turn LEDs ON (HIGH) or OFF (LOW) corresponding to the current state **(IDLE/Moving/Blocked).** The LEDs themselves (e.g., WP154A4) have specific forward voltage and current ratings (from their datasheet) that necessitate current-limiting resistors in the physical circuit (though not shown in code).

```
#include <Servo.h>

// --- Configuration ---
const int LEFT_SERVO_PIN = 3;
const int RIGHT_SERVO_PIN = 2;

// Use your calibrated STOP values!
const int LEFT_STOP_US = 1500;
const int RIGHT_STOP_US = 1510; // Example calibrated value

// Use a suitable speed offset
```

```
const int FORWARD_SPEED_OFFSET = 300; // Adjust as needed

// --- STEP DURATION (Tune this!) ---
const int STEP_DURATION_MS = 250; // How long (milliseconds) to move
forward per step

Servo servoLeft;
Servo servoRight;

void setup() {
  Serial.begin(9600); // Match Python's CAR_ARDUINO_BAUD
  pinMode(LED_BUILTIN, OUTPUT);

  servoLeft.attach(LEFT_SERVO_PIN);
  servoRight.attach(RIGHT_SERVO_PIN);

  stopMotors(); // Ensure stopped at start

  Serial.println("Boe-Bot Step Controller Ready. Waiting for 'P'...");
  digitalWrite(LED_BUILTIN, LOW);
}

void loop() {
  if (Serial.available() > 0) {
    char incomingCommand = Serial.read();

    if (incomingCommand == 'P') {
      Serial.println("Received 'P': Executing Step");
      executeStep();
    }
    // Ignore other commands for now
  }
}

void executeStep() {
  digitalWrite(LED_BUILTIN, HIGH); // Indicate moving

  // Move forward using calibrated stop values + offset
  servoLeft.writeMicroseconds(LEFT_STOP_US + FORWARD_SPEED_OFFSET);
  servoRight.writeMicroseconds(RIGHT_STOP_US - FORWARD_SPEED_OFFSET);

  delay(STEP_DURATION_MS); // Move for the specified duration
```

```
  stopMotors(); // Stop automatically after the delay
  digitalWrite(LED_BUILTIN, LOW); // Indicate stopped
  Serial.println("Step Complete.");
}

void stopMotors() {
  servoLeft.writeMicroseconds(LEFT_STOP_US);
  servoRight.writeMicroseconds(RIGHT_STOP_US);
}
```

Arduino code for boebot movement logic

**VI. System Integration and Feedback Loops**

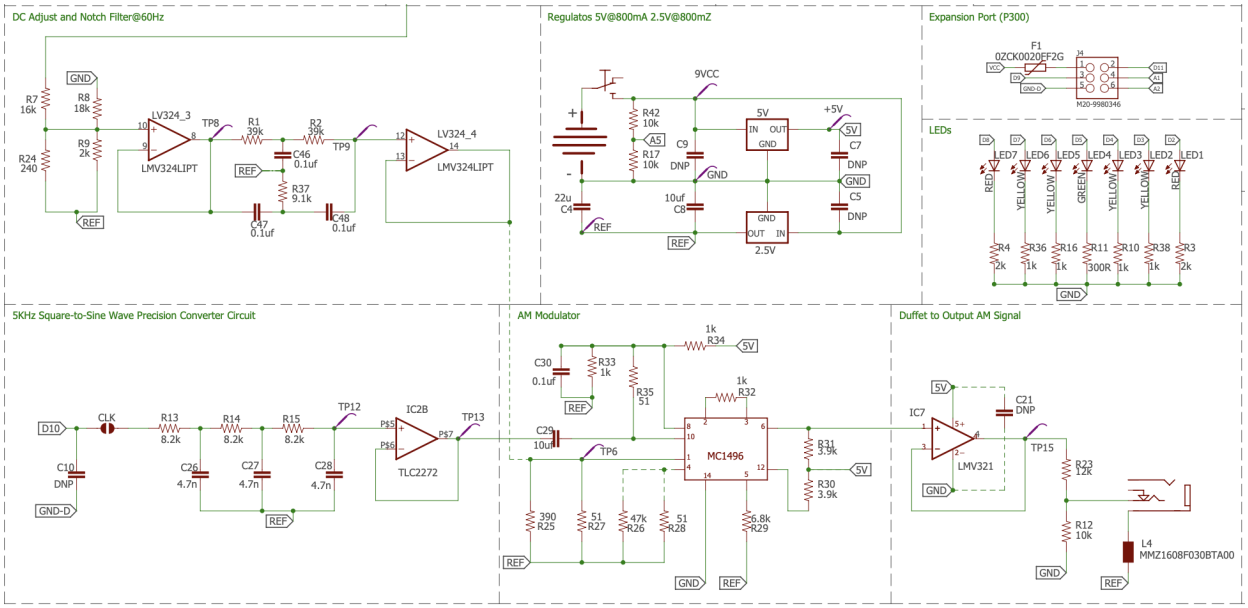
- The overall project creates multiple interacting loops

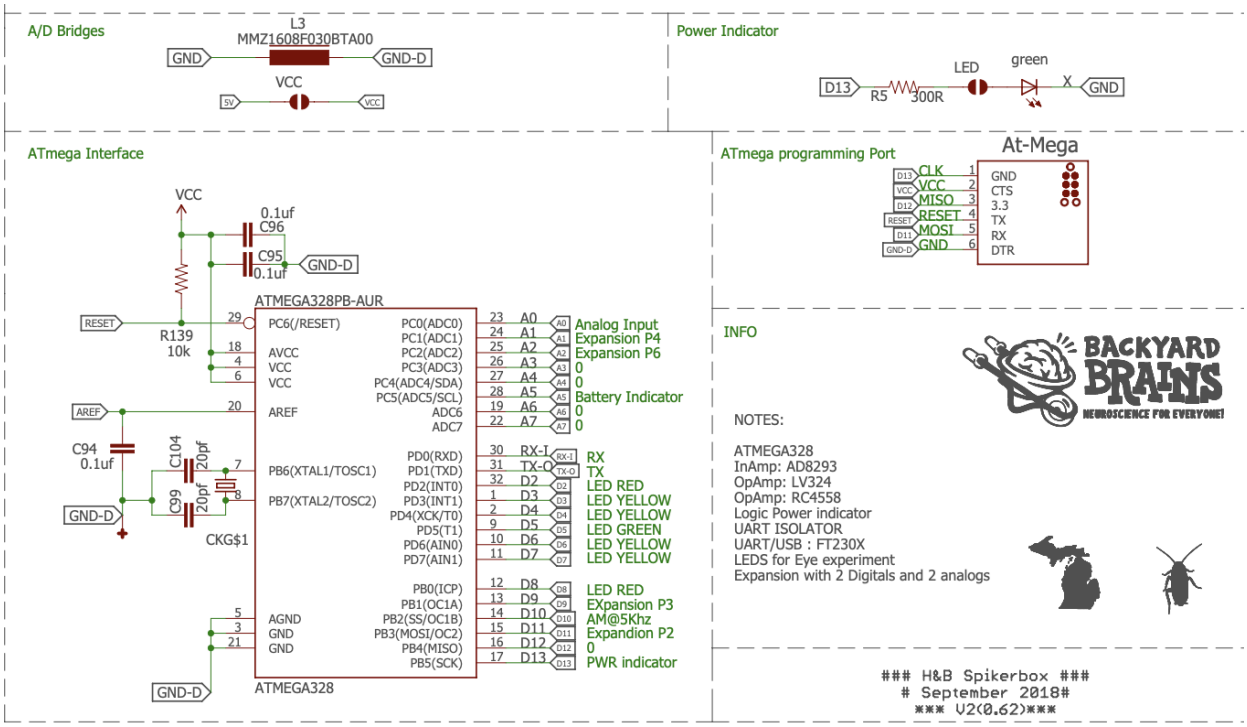
  **Primary Control Loop:** User Flex -> EMG -> SpikerBox (ADC) -> Python (Processing/FSM) -> Serial Cmd -> Arduino (FSM/Sensor Check) -> Robot Step.

  **User Feedback Loop:** User observes robot action -> User decides next flex. Environmental Feedback Loop: Obstacle -> Ultrasonic Sensor -> Arduino -> Decision Logic -> Inhibit Step Action.

  Visual Feedback Loop: Arduino State -> LED Control -> User sees robot status.

  This project provides a comprehensive practical application of converting analog signals to digital (ADC), handling binary data, using communication protocols (UART), applying combinational (comparison) and sequential logic (FSM, debouncing) for signal processing and control, integrating sensor feedback, and driving actuators and outputs based on state and input conditions.




## TEAM MANAGEMENT

This project was made possible through effective collaboration and a clear division of responsibilities within our team, “The Flexecutioners.”

- Enan focused primarily on the Python-based signal processing, including the reconstruction of EMG data, implementation of the finite state machine for step control, and the communication interface between the SpikerBox and the Boe-Bot Arduino.

•Mansour concentrated on the Arduino-side logic and actuator control, including the design of movement behaviors, LED-based state indication, and safety integration via ultrasonic sensors. He developed the microcontroller firmware that translated command signals into physical movement, ensuring safe operation with environmental awareness.
•Hardware assembly, sensor placement, and system integration were performed collaboratively, with both members contributing to debugging electrical connections, configuring sensor calibration, and aligning the system timing across hardware and software interfaces.

## Reference

For our project, we relied alot on the vast documentation provided by Backyard Brains on the spikerbox, including its inner circuit workings and its encoding principles so as to buid a working decryption for the digital signals outputted using python.


1. Backyard Brains, "Guide for USB Communication with SpikerBox," Version R7, April 5, 2024. Manufacturer Documentation. (This covers the crucial details about the USB protocol, data format, baud rate, and specific messages for your Heart & Brain SpikerBox).
2. Harvard SEAS Electrical Engineering Active Learning Labs, "Lab 9: Arduino Smart Car: the Boe-Bot," Engineering Science 50 Course Materials, Spring 2024.
3. Backyard Brains, "Heart and Brain Spiker Box Product Documentation," https://backyardbrains.com/products/heartAndBrainSpikerBox. S. Mircic (Backyard Brains), "Python script for reading, parsing and display data from BackyardBrains' serial devices," September 2019.
4. Kingbright, "WP154A4SUREQBFZGC T-1 3/4 (5mm) Full Color LED Lamp Datasheet," Rev. V.11A, March 2024
5. Arduino, "Servo Library Documentation,". Available: https://www.arduino.cc/reference/en/libraries/servo/. A
6. Arduino, "Arduino MKR Zero Documentation
7. pyserial Documentation, Chris Liechti et al., . Available: https://pyserial.readthedocs.io/.
8. Matplotlib Documentation, J. D. Hunter et al.,]. Available: https://matplotlib.org/. (The Python library used for plotting).
9. NumPy Documentation, C. R. Harris et al.,. Available: https://numpy.org/.