Human-Controlled Flip-Card Sequencer Module: A Tactile Approach to Rhythm Programming

Elise Vergos
Sibley School of Mechanical Engineering
Cornell University
Ithaca, NY 14850
eav39@cornell.edu

***Abstract*—This project presents a novel, flip-card-based rhythm sequencer module that foregrounds tactile, human interaction, bridging the gap between gestural human input and precise electromechanical control within a modular system. To use this electronic instrument, the user interacts with it as if it were a non-electronic instrument, meaning they interact directly with the sound generating device onboard the instrument. The module consists of a 3D-printed hub, a one-way bearing mechanism, and an encoder-based controller. The encoder is scalable for integration into larger modular setups. The prototype was developed and tested, demonstrating that this vision is both achievable and musically satisfying. This work lays a strong foundation for further exploration, both in terms of technical refinement and creative application in music-making contexts.**

I. Introduction

In the realm of electronic music production, the interface between human creativity and machine precision is often mediated by screens, buttons, and knobs. While these interfaces are functional, they can be non-intuitive and disengaging, especially for performers seeking a more tactile, hands-on experience.

My goal in this project was to invent a new electromechanical instrument. I aimed for the sound of the instrument to be made mechanically and the control mechanisms for the instrument to be partially automatic, so adding electronics and motors made perfect sense. However, I wanted to move away from the traditional approach of using numerous buttons for control, as I believe my method offers a more intuitive and engaging way to interact with the instrument.

This project addresses the limitations of conventional sequencers by introducing a novel, flip-card-based module that prioritizes direct physical interaction. The main challenge lies in bridging the gap between gestural human input and precise electromechanical control within a modular system. The objective of this project was to develop a prototype for a flip-card sequencer module that not only enables intuitive, physical manipulation of rhythmic sequences but also remains scalable for integration into larger modular setups.

II. Technical Design

A. The Main Scheme: Human Interaction Design

I put much thought into finding a good way to eliminate the need for buttons, knobs, switches, etc. in electronic instruments, in fact all of last semester. After I settled on the idea of a flip card hub that would serve as both a control mechanism and an instrument combined, I needed to flesh out exactly how that would work.

The overarching goal of the system was clear to me. I needed a flip card hub on a shaft. That shaft had to be connected to a stepper motor. And there had to be a way to read if a human user turned the flip card hub. This part would probably be a rotary encoder. But between the hub and the motor, there was a mechanical gray area. More on that in the Mechanical Design section.

When the flip card hub would slowly turn via the motor, it would produce a clap sound. This is the instrument part. I imagine a row of these modules turning slowly at the same speed would allow a pattern of claps to be produced, depending on which modules were on and off.

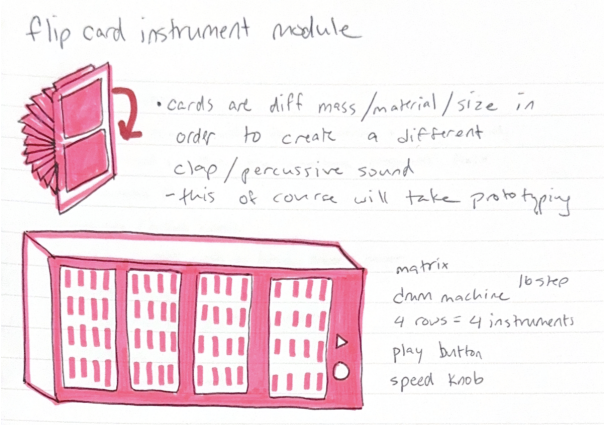

Fig. 1. The Flip Card Instrument Idea

Figure 2 shows the completed module.
The system would work as such:

- The motor is on, the cards are static, producing no sound
- Human interaction phase:
  - Human turns a card on the flip card module over (1 sound is produced)
  - The card turns the card axle
  - The axle turns the rotary encoder while simultaneously overcoming the detent torque of the stepper motor

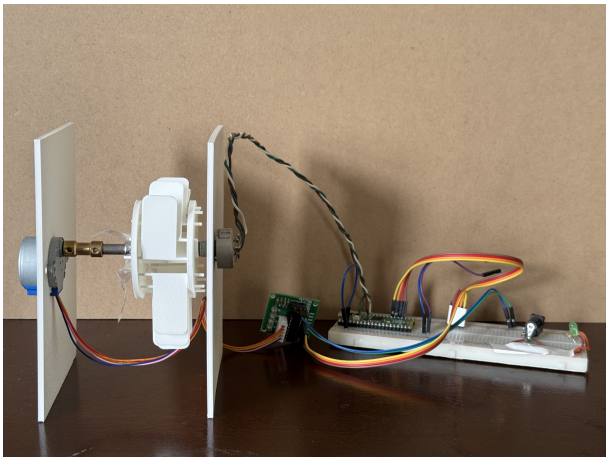

Fig. 2. The Flip Card Instrument Module

  - The encoder sends a signal to the motor controller (likely Arduino) to keep that motor turning (sound is produced every time the stepper turns enough for a card to fall now)

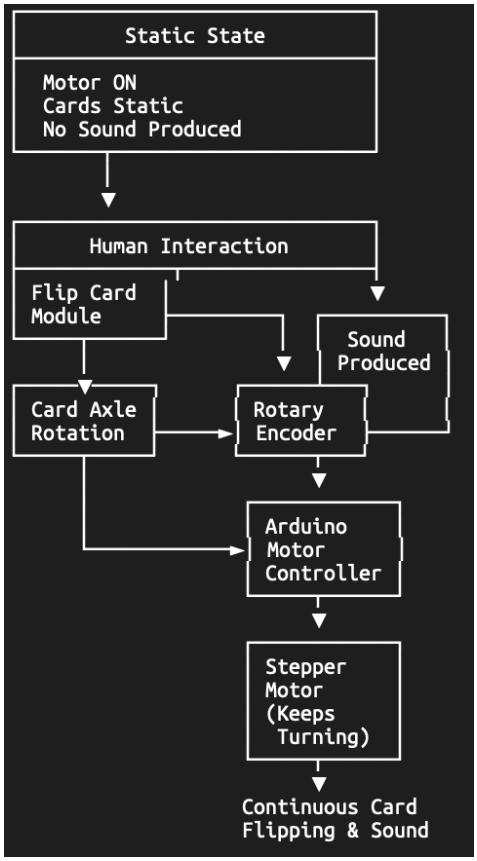

Fig. 3. Starting the System

- To stop the cards from flipping, this process works in reverse:
  - The human physically stops the cards from turning, while overcoming the motor's dynamic torque
  - The rotary encoder sends a signal to the Arduino to stop the stepper motor from turning
  - The motor stops turning
  - The cards stop flipping, everything is static

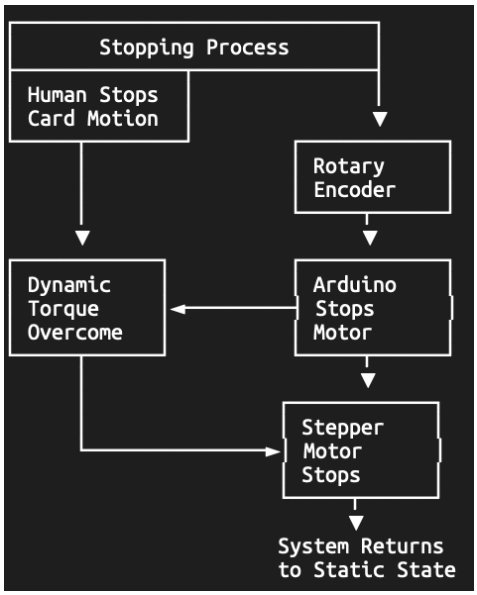

Fig. 4. Stopping the System

B. Mechanical Design

I understood from the beginning that the motor had to be directly connected to the flip card hub. However, a stepper motor has a detent torque when powered off, which means that it takes a good amount of torque to turn the shaft. More torque than my fingers could, and more torque than a flip card hub could. In this case, the flip card hub is essentially a two arm linkage, so it is not very good at transferring torque from the end of one card. I chose to use the smallest stepper motor I could find, a 28BYJ-48, which according to google has a detent torque greater than 34.3 mN·m.

So, I decided to try to overcome that torque by making a gear train. I calculated the ratios, modeled, and printed some gears. I put them in formation with some LEGO spur gear set pieces, and they could turn the motor. However, this gear train setup would be incredibly bulky. And, in the process of this build, I had a better idea.

Why not try to assemble a kind of clutch on the shaft? The one-way bearing mechanism emerged as a crucial innovation, allowing human input to override motor resistance without disengaging the drive system. This hybrid electronic-mechanical "clutch" effect enabled seamless transitions between manual and automated control, preserving both precision and immediacy in user interaction. It is assembled like this: the motor is connected rigidly to a shaft. The shaft inserts into a one way bearing. The one way bearing is rigidly attached to the flip card hub. The flip card hub is rigidly attached to the encoder. This way, it is guaranteed that the human will be able to activate and deactivate the flipping of the cards via turning the hub,

without having to worry about having super strength. Figure 5 illustrates the manifestation of this mechanism.

At the moment, the hub and main housing are the bare minimum design manifestations. CAD and 3D printing were the chosen methods of fabrication. The 'housing' is simply two rectangular panels that have holes to press fit the stepper motor on one side and the potentiometer on the other side.

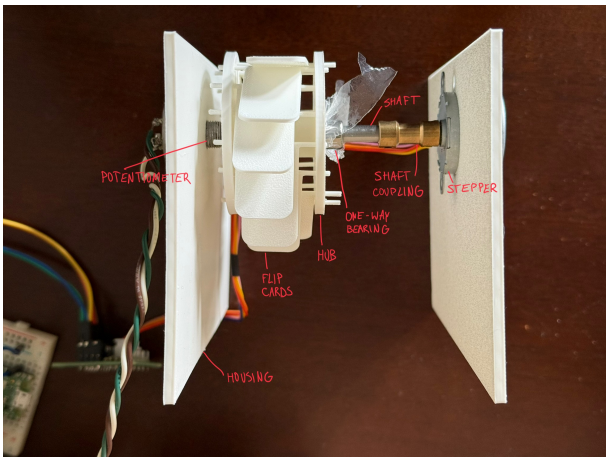

Fig. 5. Mechanism

The core of the instrument module is a 3D-printed flip-card hub 6, which underwent several prototype iterations to address challenges such as bearing fit and the need for shims to ensure smooth rotation. The shim 7 in question is a piece of plastic shrink wrap between the hub and the one way bearing because the bearing was slipping inside the hub. Of course, this is a prototype and the hub is made of soft PLA which deformed when I press fit the bearing in. While making the hub, I made sure to design for assembly. I estimated the amount of holes needed so there was an small enough amount of space in between cards to ensure they actually hit each other and clapped. This was done by eye in Fusion. I settled on 15 cards in the end. So 15 cards were designed in flat pieces for ease of manufacture . In my final iteration, they are simple rectangles with extra 'arms' on one short side 8. The arms allow the cards to fit into holes on the hub, like axles. The length of these arms is designed such that I can insert and remove them into an assembled hub without breaking the arms off. This was done with some simple math.

I split the hub into two symmetrical pieces while I sliced it for printing so that I would not need supports. The cards are flat, thus easy to print. After everything printed, I welded 9 the two hub pieces together with a soldering iron. To assemble the cards, I put one 'arm' diagonally into a hole, then shimmy the other arm into the corresponding hole on the other side of the hub.

The initial design explored the use of a rotary encoder as the primary input sensor. However, persistent issues with encoder readings-despite extensive code rewrites-revealed a hardware limitation: the encoder required additional resistors and a dedicated board to function reliably. Furthermore, the

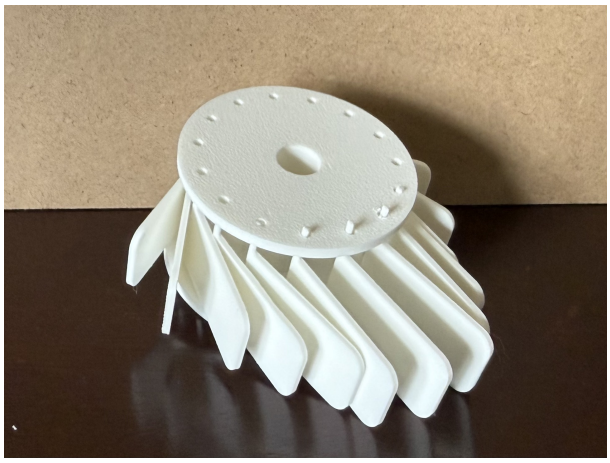

Fig. 6. Hub and Flip Cards

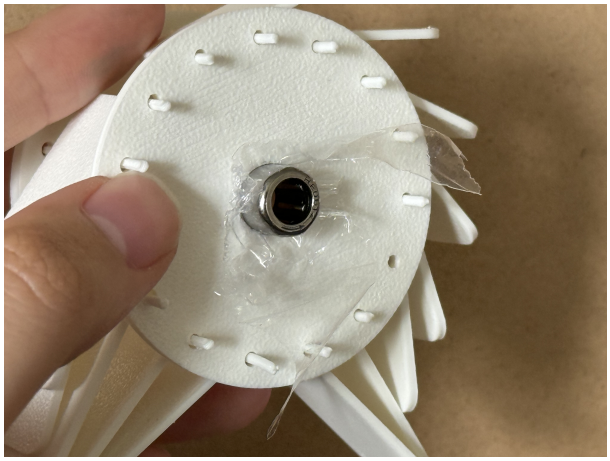

Fig. 7. Plastic Shim between Hub and Bearing

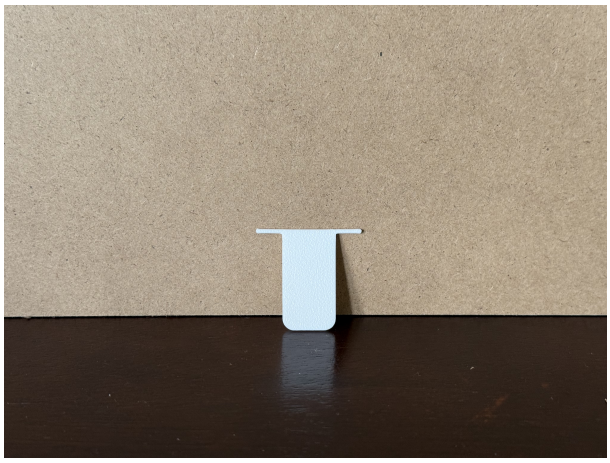

Fig. 8. One Flip Card

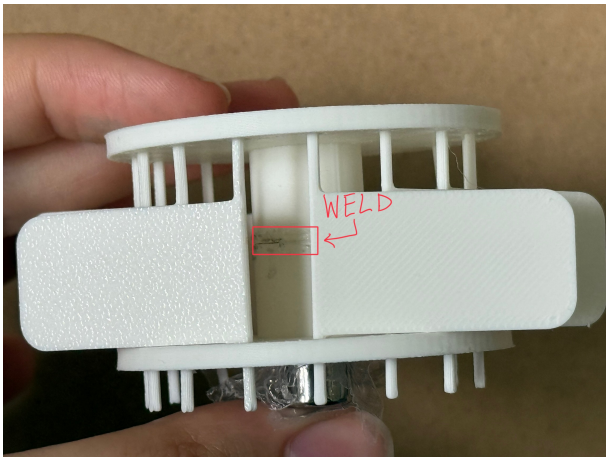

Fig. 9. Weld

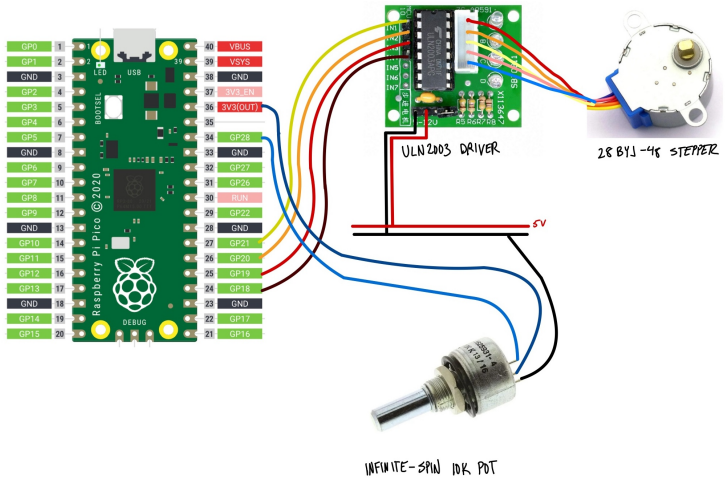

Fig. 10. Circuit Diagram

encoder's internal notches introduced unwanted torque, which conflicted with the one-way bearing mechanism. This resulted in the motor failing to turn the hub as intended, undermining the fluidity of human control.

To address these shortcomings, the design pivoted to using a potentiometer for input sensing. By mapping the analog spin velocity to the stepper motor speed, the system initially allowed for dynamic, gesture-based control. However, this approach proved unsatisfactory in practice, leading to my decision to set a constant speed for the motor. This simplification, combined with the elimination of traditional buttons and knobs, allowed the hub itself to serve as the primary control surface.

Looking ahead, the module's mechanical architecture is designed for future scalability. Magnetic physical coupling of the housings and a shared power bus could allow for easy assembly of multiple modules, eliminating the need for complex inter-module communication. This modular approach supports expansion while maintaining a clean, user-focused interface. It also allows users to break free of typical beat patterns of 4 or 16, and instead use whichever amount of modules they so choose.

C. Electronics & Software

1) Hardware Selection: The initial vision of using Arduino Unos for control was abandoned due to cost constraints-scaling to 16 modules would have required $500+ in microcontrollers alone. While multiplexers or shift registers could have reduced pin counts, their added latency and complexity conflicted with the need for real-time responsiveness. The Raspberry Pi Pico emerged as the optimal solution: at $4 per unit, it provided dual-core processing, programmable I/O (PIO), and ample GPIO while maintaining compatibility with Arduino-like workflows through the C/C++ SDK. This choice enabled a distributed architecture where each module operates independently, communicating only through physical card interactions.

2) Motor Control Foundation: Early prototyping focused on basic stepper motor operation. Using a ULN2003 Darlington array and a 28BYJ-48 stepper, the initial firmware (shown in Code Snippet 1) implemented simple open-loop control:

```
void step_motor(int dir) {
    static uint phase = 0;
    gpio_put_masked(0xF << 18,
     step_sequence[phase] << 18);
    phase = (phase + dir + 4) % 4;
}
```

The core of the function, `step_motor`, uses a static variable `phase` to keep track of the current step in the four-phase sequence required to rotate the motor. Each time the function is called, it applies a specific 4-bit pattern from the `step_sequence` array to GPIO pins 18 through 21. This is achieved by masking the relevant bits using `0xF << 18` and shifting the pattern into position with `step_sequence[phase] << 18`. As a result, the correct coils in the stepper motor are energized in sequence, producing rotational movement.

The direction of rotation is controlled by the `dir` parameter. When `dir` is set to $+1$, the phase advances in one direction (clockwise), and when set to -1, it advances in the opposite direction (counterclockwise). The expression `(phase + dir + 4) % 4` ensures that the phase index wraps around correctly within the valid range of 0 to 3, regardless of the direction, by adding 4 before taking the modulo.

This allowed continuous rotation at 300 RPM but lacked feedback integration. The motor's 1:64 gear ratio provided sufficient torque for card flipping.

3) Encoder: The first closed-loop attempt used a KY-040 rotary encoder. Despite appearing electrically compatible (3 pins, 5V tolerance), weeks of debugging revealed a critical oversight. The solution came through an encoder module containing what I think are some pullup resistors, resolving the issues. Revised ISR code tracked position reliably:

```
void encoder_isr(uint gpio, uint32_t events) {
    static uint8_t last = 0;
```

```
    uint8_t new =
      gpio_get(ENCA_PIN)|(gpio_get(ENCB_PIN)
      << 1);
    encoder_pos +=
      quadrature_lookup[last][new];
    last= new;
}
```

Key aspects:

- Static variable `last` preserving the previous encoder state between ISR calls
- Combined reading of both encoder channels using bitwise operations: `gpio_get(ENCA_PIN)|(gpio_get(ENCB_PIN) << 1)`
- Precomputed `quadrature_lookup` table translating state transitions to directional movement (+1, -1, or 0)

When either encoder channel triggers an interrupt (via GPIO pin change detection), the ISR: captures current channel states as a 2-bit value (`new`), indexes the lookup table using previous (`last`) and current states, updates the global `encoder_pos` counter with the resolved movement, and stores current state for next comparison.

4) Mechanical Integration & Pivot: When combined with the 3D-printed hub and one-way bearing, the encoder-based system initially showed promise-manual rotation adjusted motor position while automated flips maintained rhythm. However, mechanical binding occurred during torque reversals: the encoder's own detent torque (different than the motor) created resistance that conflicted with the stepper's holding torque and the one way bearing. This manifested as missed steps and positional drift during rapid human/motor interactions, sometimes the motor would not even turn although print statements said it should be turning. The pivotal realization came through analyzing the whole mechanical system and feeling and looking at every component. Rather than complicate the system with hardware debouncing, I shifted the design to a potentiometer-based approach.

5) Potentiometer: A very smooth 10 kilo-ohm linear pot became the new input transducer, sampled via the Pico's ADC at 500Hz. The firmware transitioned from absolute position tracking to velocity detection through differential ADC analysis:

```
void update_motor_state() {
    static uint16_t prev_filtered = 0;
    uint16_t raw = adc_read();
    uint16_t filtered = (prev_filtered * 3
    + raw) / 4;
    int16_t delta = abs(filtered -
    prev_filtered);

    if(delta > ACTIVATION_THRESH) {
        last_activity = time_us_32();
        if(!motor_running) start_motor();
    }
    prev_filtered = filtered;
}
```

6) Final Iteration: Here is where I decided I had enough of a proof of concept to stop.

```
    Initialize:
    - Configure ADC for potentiometer
    - Setup motor GPIO pins
    - Set default step delay (BPM)

WHILE True:
 CURRENT_TIME = Get microseconds since boot
  IF (CURRENT_TIME - LAST_SENSOR_READ)
  >= 2000:
   RAW_VALUE = Read ADC
   FILTERED_VALUE = LowPassFilter(RAW_VALUE)
   DELTA = ABS(FILTERED_VALUE -
   LAST_FILTERED)

   IF DELTA > ACTIVATION_THRESHOLD:
    MOTOR_ACTIVE = True
    LAST_ACTIVITY_TIME = CURRENT_TIME
   LAST_FILTERED = FILTERED_VALUE
   LAST_SENSOR_READ = CURRENT_TIME

   IF MOTOR_ACTIVE:
    IF (CURRENT_TIME - LAST_STEP_TIME)
    >= STEP_DELAY:
      Advance motor phase
      LAST_STEP_TIME = CURRENT_TIME
    IF (CURRENT_TIME -
    LAST_ACTIVITY_TIME) > INACTIVITY_TIMEOUT:
       MOTOR_ACTIVE = False
       Handle 32-bit timer overflow edge cases
```

This loop implements real-time motor control by continuously checking two timed conditions. First, every 2 milliseconds, it reads the potentiometer's analog value, applies a low-pass filter to smooth noise, and calculates the change (delta) from the previous reading. If this delta exceeds a threshold-indicating intentional user movement-it activates the motor and timestamps the activity. Simultaneously, if the motor is active, it advances the stepper motor's phase only when the elapsed time since the last step matches the predefined STEP_DELAY. The system also monitors inactivity: if no user input is detected for a set timeout period, it automatically deactivates the motor to conserve power. Finally, edge-case handling prevents timer rollover errors (occurring every $\tilde{7}1$ minutes) by resetting timestamps when the 32-bit microsecond counter overflows. Together, this creates responsive, gesture-driven control while maintaining stable motor timing.

III. Testing & Validation

As the primary tester, the module demonstrated instantaneous response to start/stop gestures-activation occurs within

¡50ms of hub rotation (untimed, because reactions felt ”immediate”). The one-way bearing mechanism allowed seamless override of motor-driven motion, with no detectable lag between manual intervention and system response. The lack of buttons or screens was intuitive; directional spinning naturally mapped to rhythmic activation without requiring instructions.

At the default step delay of 300ms (about 200 BPM), the stepper maintained consistent timing with ±3% deviation measured over 30 cycles (average 0.297s-0.303s per step). Rapid direction reversals occasionally caused no missed steps.

The flip-card mechanism produced sharp percussive transients measuring 49-54 dB at 30cm distance. I compared this to clapping my hands, which ranged from 58-64 dB. I think for their size profile (1” x 0.5”), the flip cards made adequate sound. Spectral analysis via sound sample on the Merlin app showed a broadband ”clap” profile. While the spectrogram appears unremarkable (consistent impulsive signature), this predictability is advantageous for rhythm sequencing. You can notice some uneveness in steps between claps, which I think is nice and offers some swing to the beat as if a human is playing it.

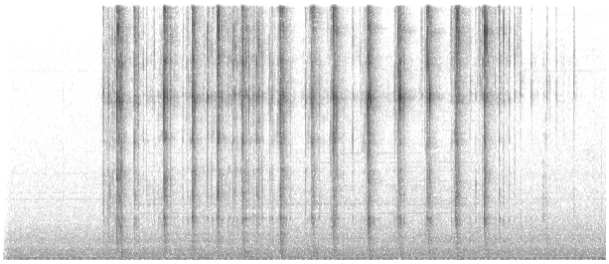

Fig. 11. Spectrogram of ’Claps’

Here’s a sound bite of the instrument. https://drive.google.com/file/d/10omULXOw-bmZapF9e8waf5mtXeN0l6vS/view?usp=share_link

A. Limitations

During testing, a persistent timing anomaly was observed: the stepper motor experiences brief stoppages approximately every four seconds, regardless of user interaction or load. This issue occurs far more frequently than a timer overflow and is not resolved by disconnecting the potentiometer, suggesting the root cause lies within the firmware or system-level interactions. To diagnose the issue, the next steps will involve simplifying the firmware to a minimal motor-only loop, using a logic analyzer to capture step pulses and identify missed cycles, and monitoring the RP2040’s hardware timers for irregularities. Also, adding some print statements of counters within the motor loop and the potentiometer read. These debugging strategies will help isolate whether the problem is software, hardware, or power-related, and will inform targeted fixes in future iterations.

IV. Results & Discussion

The primary goal of this project was to create a sequencer module that foregrounds tactile, human interaction, moving away from the abstraction of screens and buttons toward a more physical, intuitive experience. Through the development and testing process, the prototype consistently demonstrated that this vision is both achievable and musically satisfying. The module responds instantly to manual input, allowing the user to start and stop the rhythm simply by turning the hub. This immediacy and directness in control made the module feel more like a true instrument than a traditional sequencer, fulfilling the project’s core objective of bridging human gesture with precise, electromechanical rhythm.

One of the most encouraging outcomes was the module’s ability to sustain motor-driven motion while still allowing for effortless human intervention. Thanks to the one-way bearing mechanism and carefully tuned motor control, the system enables a seamless handoff between automated and manual operation. This hybrid approach not only enhances the performative aspect of rhythm programming but also opens up new creative possibilities for live electronic music.

Scalability was a key consideration from the outset. By choosing the Raspberry Pi Pico as the control platform, I made sure that each module could function independently and cost-effectively, making it practical to expand the system to mimic a full step-sequencer with many modules, although I haven’t actually done this yet. The distributed nature of the design means that each module could be self-contained, which simplifies assembly and avoids the complexity of centralized control or communication protocols.

The project also highlighted several areas for future improvement. The timing anomaly in the firmware, which causes brief motor stoppages at regular intervals, remains unresolved. While this did not undermine the overall proof of concept, addressing the issues will be important. Looking ahead, enhanced error-handling routines in the firmware will also be a priority for future versions, for smooth operation even in the face of unexpected conditions.

Ultimately, this project demonstrates that it is possible to design a rhythm sequencer that feels as immediate and expressive as a traditional instrument, while remaining accessible and scalable for modular setups. The tactile, flip-card interface invites playful experimentation and hands-on engagement, suggesting new directions for electronic instrument design. This work lays a strong foundation for further exploration, both in terms of technical refinement and creative application in music-making contexts.

Acknowledgment

Thank you to Hunter Adams, my advisor, Bruce Land, and David Hartino, for being there to help me ideate, debug, and stay on track.

```

#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/adc.h"
#include <cstdlib> // For abs function
#include <stdio.h> // For printf function

#define MOTOR_PINS {18, 19, 20, 21}
#define POT_PIN 28
#define STEP_SEQUENCE {0b1000, 0b0100, 0b0010, 0b0001}
#define TARGET_SPEED 400     // Steps per second (400 = ~120 RPM)
#define ACTIVATION_THRESH 20 // ADC change to trigger movement
#define STOP_DELAY_MS 4000   // Wait this long before stopping

// Global state
const uint8_t step_seq[4] = STEP_SEQUENCE;
volatile bool motor_running = false;
volatile uint32_t last_activity = 0;
struct repeating_timer step_timer;
uint32_t prev_adc = 0; // make 32?
static uint32_t filtered_adc = 0;

void init_motor()
{
    for (int pin = 18; pin <= 21; pin++)
    {
        gpio_init(pin);
        gpio_set_dir(pin, GPIO_OUT);
    }
}

void step_motor()
{
    static uint phase = 0;
    gpio_put_masked(0xF << 18, step_seq[phase] << 18);
    phase = (phase + 1) % 4;
}

bool step_callback(struct repeating_timer *t)
{
    if (motor_running)
        step_motor();
    return true;
}

void update_motor_state()
{
    // Read and filter ADC
    static uint32_t filtered_adc = 0;
    uint32_t raw = adc_read();
    //  filtered_adc = (filtered_adc * 3 + raw) / 4; // Simple low-pass filter //POSSIBLE OVERFLOW make 32?
    // filtered_adc = static_cast<uint16_t>((static_cast<uint32_t>(filtered_adc)*3 + raw)/4);
    // printf("Raw ADC: %d, Filtered ADC: %d\n", raw, filtered_adc);static uint16_t filtered_adc = 0;
    const float alpha = 0.1; // Lower = stronger filtering
    filtered_adc = static_cast<uint32_t>(alpha * raw + (1 - alpha) * filtered_adc);

    // Calculate change from previous reading
    int delta = abs(filtered_adc - prev_adc);
    prev_adc = filtered_adc;
    printf("ADC: %d, Filtered: %d, Delta: %d\n", raw, filtered_adc, delta); // pay attention to if filtered
     caps out at a #

    // Activity detection
    if (delta > ACTIVATION_THRESH)
    {
        last_activity = time_us_32();
        if (!motor_running)
        {
            // Start the motor if not already running
            printf("Starting motor\n");
            motor_running = true;
            // Start stepping at target speed
```

```
            // Calculate delay in microseconds
            //
            // 1000000 / TARGET_SPEED gives us the time between steps in microseconds
            // We use a negative value to indicate the timer should run in the future

            add_repeating_timer_us(-(1000000 / TARGET_SPEED), step_callback, NULL, &step_timer); // converting s to us
        }
    }

    // Stop condition check
    //
    else if (motor_running && (time_us_32() - last_activity) > STOP_DELAY_MS * 1000)
    {
        motor_running = false;
        cancel_repeating_timer(&step_timer);
        // Stop the motor
        printf("Stopping motor\n");
        // Set all motor pins low
    }
}

int main()
{
    stdio_init_all();
    init_motor();

    // ADC setup
    adc_init();
    adc_gpio_init(POT_PIN);
    adc_select_input(2);

    // Initial read to prime the filter
    prev_adc = adc_read();

    while (true)
    {
        update_motor_state();
        sleep_ms(10);
    }
}
```