Perpetual Motion Machine

Introduction

The perpetual motion machine is a kinetic sculpture that simulates the concept of perpetual motion. In reality, our system uses an inductive proximity sensor and an electromagnet to sense and propel the ball back to the funnel. We are assuming that the user is anyone who enjoys trinkets. As a result, we want our perpetual motion machine to function consistently. In order to enjoy the sculpture, the user will need access to a two-prong outlet with power so that they may turn the machine on. They will also need to be able to place it on a flat surface so that the ball bearing can properly return to the funnel.

Requirements

R1: The system shall toggle between on and off when a user asks Alexa.
R2: When the inductive proximity sensor senses a ball, the system shall send a PWM value of 0 to the electromagnet.

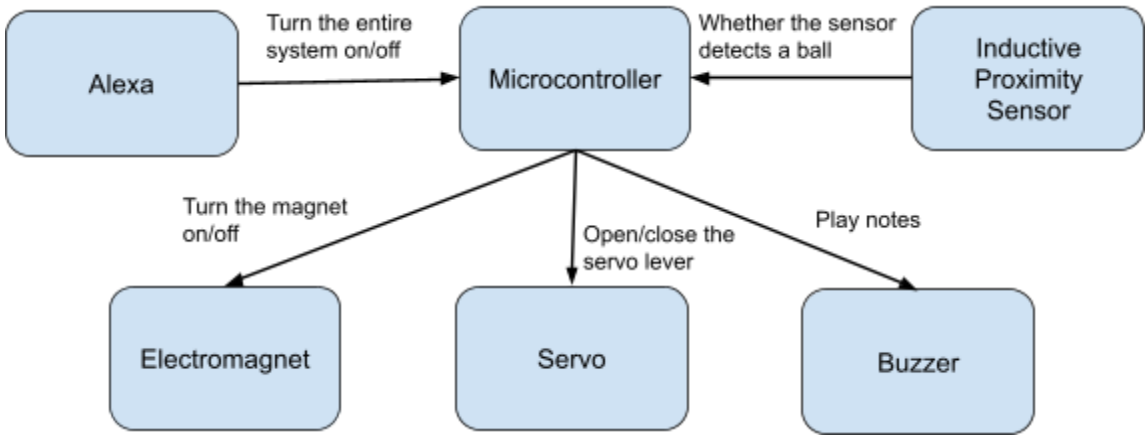
- Our system uses a P-channel mosfet to control how much power the electromagnet receives. Thus, 0 represents turning the magnet on while 255 represents turning the magnet off. We can use values between 0 and 255 to adjust the electromagnet's power.

R3: When the electromagnet has been on for 16 milliseconds, the system shall send a PWM value of 255 to the electromagnet.

- Our testing has shown that 16 milliseconds is enough time to give the ball enough kinetic energy to clear the ramp and make it back to the initial platform.

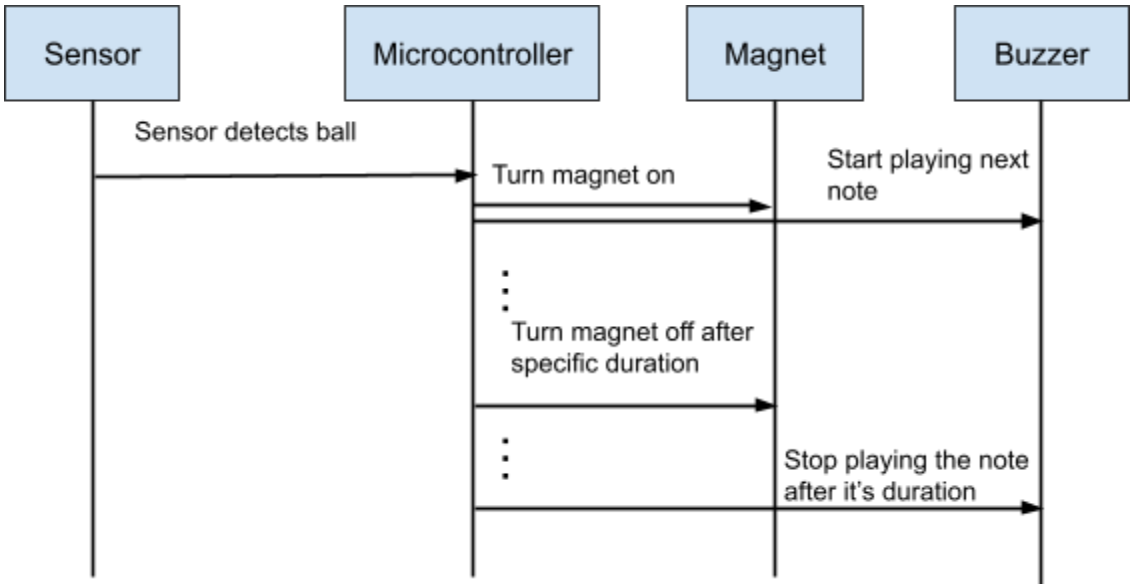
R4: When the ball has been sensed, the next note in the melody sequence shall be played.

Architecture Diagram

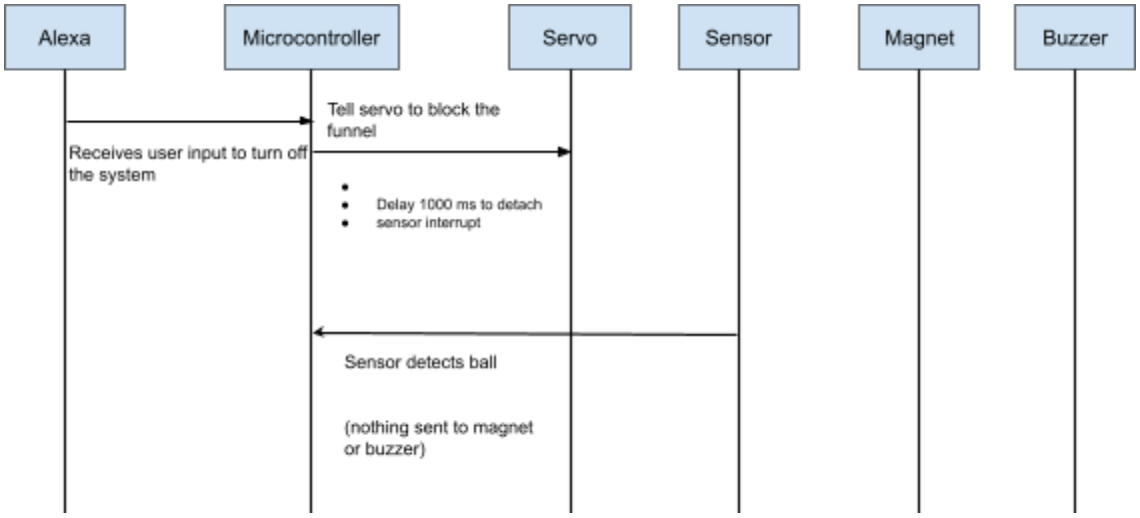


Sequence Diagrams

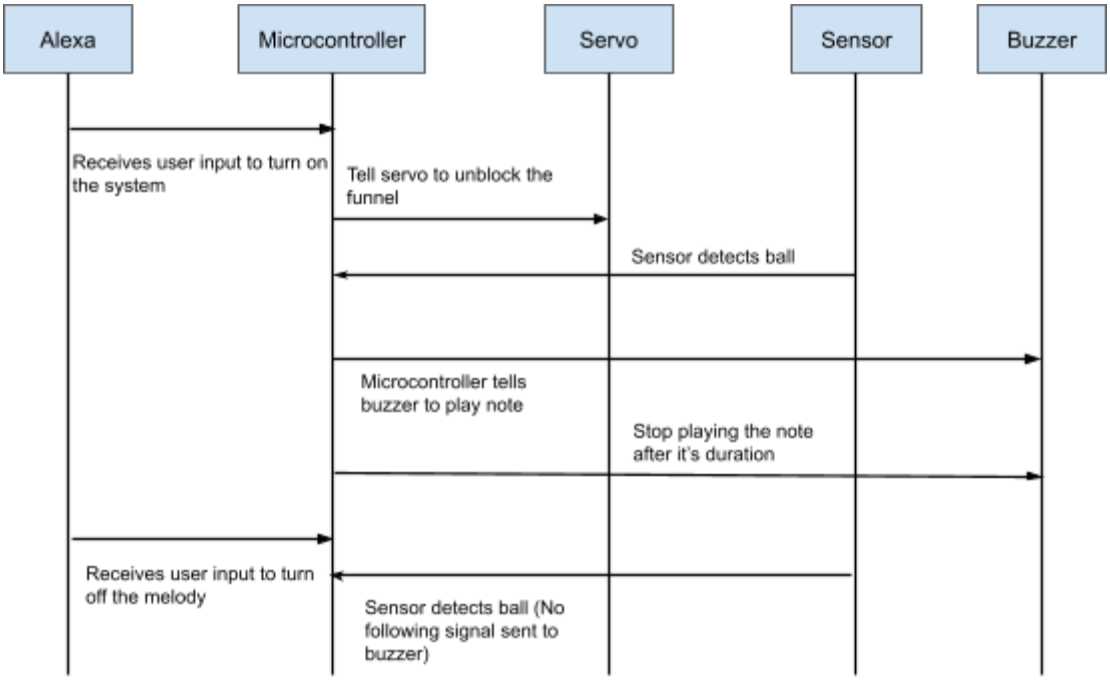
1. After the system starts a ball is placed on the ramp and rolls down, the sensor detects a ball and turns the magnet on for a period of time.



2. The system is currently on, and the user tells Alexa to turn the system off.



3. The system is currently off, and the user wants to turn the machine on then shortly afterwards turn the music off.



Finite State Machine

Constants:

- MAGNET_STRENGTH - number, how strong the magnet should be
- MAGNET_DURATION - number, how many ms the magnet should stay on for
- notes - number array, the melody to be played by the buzzer

Inputs:

- isOn - boolean that determines whether the entire system should be on or off
- mils - current clock time in milliseconds
- ballSensed - boolean that reflects if the inductive sensor detected a ball (T == detected; F == no detection)

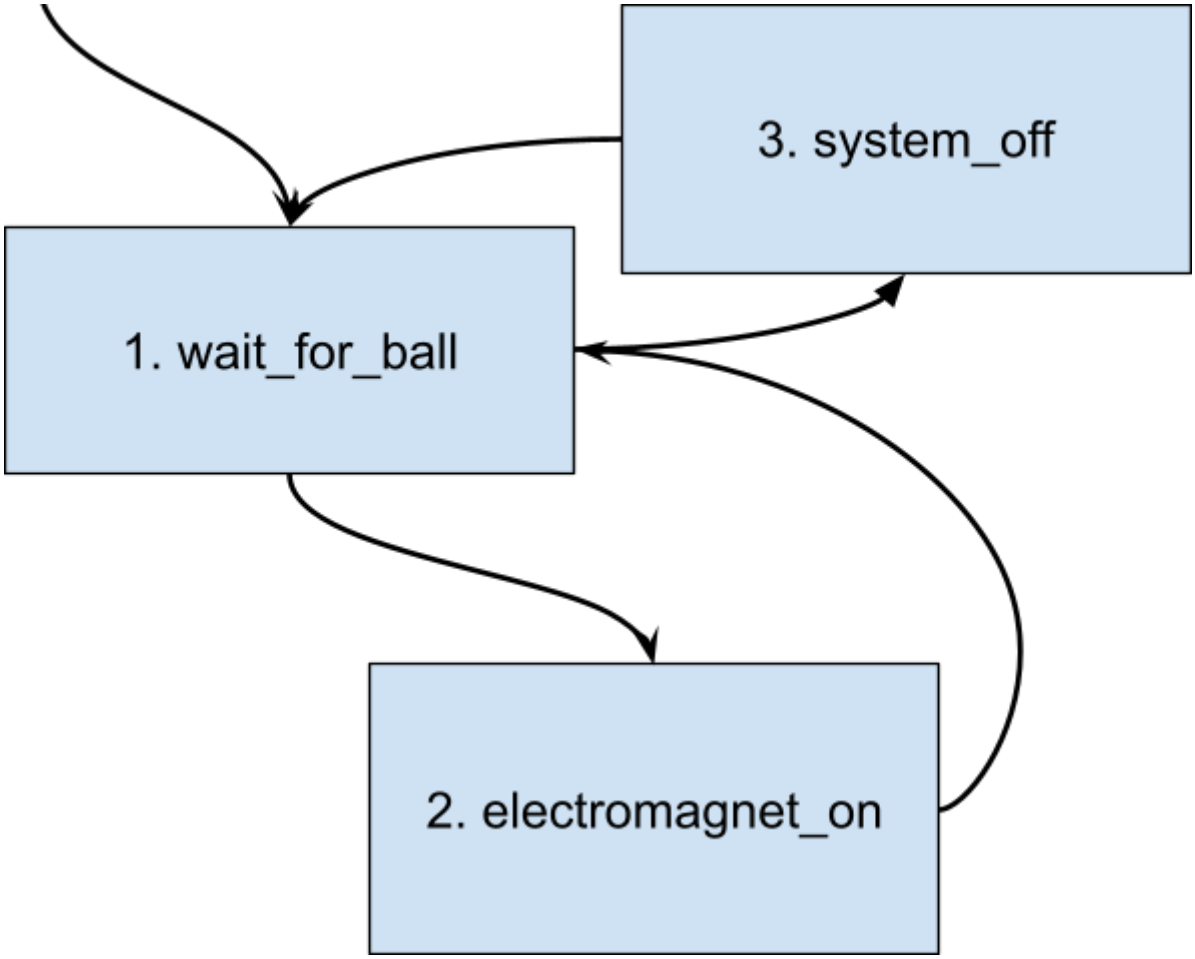
Outputs:

- toggleMagnet(bool on) - toggles the magnet either on (T) or off (F)
- playNote() - plays note on the buzzer, where ballCount determines the index from the notes array to be played.

Variables

- savedClock - saved value of clock.
    - Initializes to mils
- ballCount - number of times the ball has looped. Resets to zero once it has reached the size of the notes array.
    - Initializes to zero

Initial state is 1 (wait_for_ball).
Note: our FSM has implicit self-transitions if no other transition is taken.



Transition	Guard	Explanation	Output	Variables
1-2	ballSensed ^ isOn	Ball is sensed and ISR is triggered	toggleMagnet(true) playNote()	savedClock := mils ballCount += 1
2-1	mils - savedClock >= MAGNET_DURATION	Enough time has passed, turn the magnet off	toggleMagnet(false)	
1-3	¬isOn	System is deactivated and lever goes down		
3-1	isOn	System is activated and lever goes up		ballCount := 0

Traceability Matrix

	R1	R2	R3	R4
S 1	X		X	
S 2		X		X
S 4	X			
T 1-2		X		X
T 2-1			X	
T 1-3	X			
T 3-1	X			

Testing Approach

We will use a combination of automatic and manual testing. For automatic testing, we can unit test every transition because our finite state machine does not have a large number of states and transitions. Similar to what we did in lab 6, we can test our transitions by giving our FSM initial parameters such as isOn, sensorFlag, and mils, then asserting that our end state values are as expected. Our main coverage metric would be transition coverage, and we would only be satisfied if we reach 100% coverage in this aspect. We can also use our sequence diagrams to evaluate if we have written enough automated tests by ensuring that we write tests for each sequence. Furthermore, we can use black box testing to test our updateFSM function by evaluating when the conditionals are true/false. We can determine if we have tested enough if we have achieved full branch coverage of our updateFSM() function and output functions (like playNote() and toggleMagnet()).

For system testing, we opt to use a more manual approach, where we can run the machine for some time with the ball bearing. If the ball bearing is successfully launched back to the funnel continuously, then we know all parts of our FSM are working. We can also check the condition where we let the machine run for some time, remove the ball bearing, then allow it to run for more time. We can also test that our sensor and magnet are being properly triggered by placing the ball bearing directly over the sensor and seeing if the magnet is turned on and off properly. Our testing will be sufficient if our machine can run interrupted for an extended amount of time.

Safety and Liveness Requirements

Safety requirements:

1. ¬(¬isOn ∧ toggleMagnet(true))
    a. The machine must not turn the magnet on when the entire system is off.

2. ¬(¬ballSensed ∧ playNote())
    a. The machine must not play a note if a ball is not sensed.

Liveness Requirements

1. G[isOn -> F(wait_for_ball)]
    a. If the machine is on, it will eventually be in the “wait_for_ball” state.

2. G[(isOn ^ ballSensed) -> F(electromagnet_on ^ toggleMagnet(true))]
    a. If the machine is on and a ball is sensed, then it will eventually turn the electromagnet on.

3. G[(isOn ^ ballSensed ^ musicEnabled) -> F(playNote())]
    a. If the system is on, a ball is sensed, and music is enabled, then a note must eventually be played.

Environment Processes

One of the environmental processes that we would model is the physics of the ball as it moves through our ramp. We would likely model these mechanisms as a continuous and deterministic system. It is continuous since the ball’s position and velocity are continuously changing as it moves through our system as a result transfers of energy (gravitational potential energy to kinetic energy, interactions with our electromagnet’s magnetic field, etc.). There aren’t any jumps besides when the electromagnet turns on (but the strength of the electromagnet can be an input). It is deterministic because in an ideal model, there isn’t any randomness involved. The laws of classical physics that we are using should be deterministic.

Another process we could model is the strength of the electromagnet. This would likely be a discrete and non-deterministic system. It is discrete because the electromagnet is either on or off, and it is non-deterministic because it can switch between on and off at arbitrary moments (whenever a ball passes the sensor). There is also an argument to be made for a small continuous component as well because of the discharge of the battery/capacitors to the electromagnet, but we could reasonably make the simplification that we don’t need this level of detail in our model.

Telling Alexa to toggle the system could also be modeled as a discrete non-deterministic system since it similarly can happen at any moment and is a discrete event.

For our purposes, time can be modeled as a discrete and deterministic system. It is discrete because we are only counting each millisecond, and it is deterministic because it advances at a constant rate.

Code Description

File descriptions

- thingProperties.h contains the function declarations required to connect to Alexa.
- perpetual-motion.ino contains the core functionality, including setup and management of the watchdog timer, melody, ball count, and magnet toggling via interrupt. Our unit tests can also be found within this file.
- arduino_secrets.h defines the variables that store the network information.
- sketch.json is a file that was generated by the Cloud IDE.
- The lib folder and the files it holds are library packages that define pitches to be played on the piezo speaker.

Requirements trace

- Use PWM, ADC, or DAC
    - PWM used on perpetual-motion.ino, lines 184 and 187
- Have a watchdog timer
    - perpetual-motion.ino, lines 83, 94-118, 144
- Have at least one interrupt service routine (watchdog timer doesn't count)
    - perpetual-motion.ino, lines 77, 121-135, 223, and 228
- WiFi
    - perpetual-motion.ino lines 71, all arduino_secrets.h, thingProperties.h lines 3-19

How to run Unit Tests

The tests are run similar to the method for lab 6. Instead of using a script to generate our tests, however, we manually wrote start and end state variables. testUpdateFSM() passes these start conditions into our updateFSM function, and the tests assert the end states are as expected. To run tests, remove the code from loop(), comment out the watchdog, and call testUpdateFSM() in setup().

Note: the version of code submitted to gradescope is already set up for testing with normal functionality commented out - uncomment those parts if you want normal functionality.

Reflection

At the start of the project we set out to make a perpetual motion machine that would appear similar to the one analyzed in this video. We met this goal. Our machine is able to drop a ball through the funnel and accelerate it enough using the electromagnet to make it land back into the funnel. Our biggest challenge for this project was generating enough power for the magnet. At first, we used a 9 volt battery to charge our capacitors, but found we couldn't generate enough power. We ended up using a wall plug and a step-up converter to increase our voltage. We also had problems with the ball bouncing out of the funnel, which we solved by going through several iterations of the funnel, including extruding a spline to slow the ball down and give the capacitors time to charge. We also had to find another mosfet that could handle the

increase in voltage. Originally, we wanted to build our own electromagnet, but we also could not source adequate materials from the BDW. Instead, we purchased one from Amazon.

FSM review spreadsheets: 1, 2, 3