CSCI 1600 Final Project Report: Zen Garden Timer

Divyam Dang, Dylan Brady, Jian Cong Loh, Sahil Bansal

1. Introduction

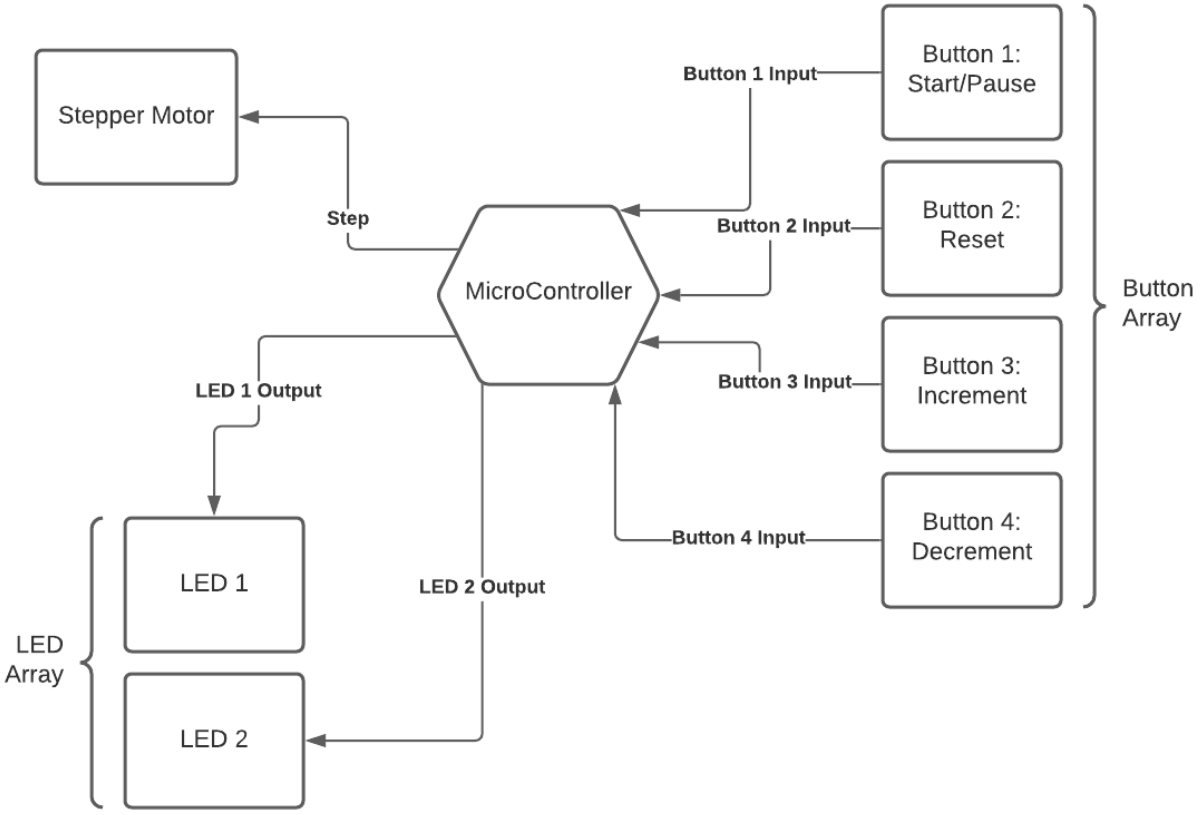
Our project is a timer that visually measures the passage of time, taking inspiration from sand-drawing zen gardens. There is a ball magnet on a bed of sand at the start of a track. When the timer is started, the ball magnet moves across the track of sand to the other end. The distance that the ball magnet has traveled relative to the total distance of the track corresponds to the amount of time that has passed relative to the timer duration. The timer allows for a 1, 2, 3, and 4 minute time interval and can be paused at any given time. The timer can also be reset after it is paused or has finished.

We assume that the user is familiar with how timers are normally operated and does not interact with the object in any way other than via the four provided buttons. In particular, we assume that the user does not tamper directly with the ball magnet on the track.

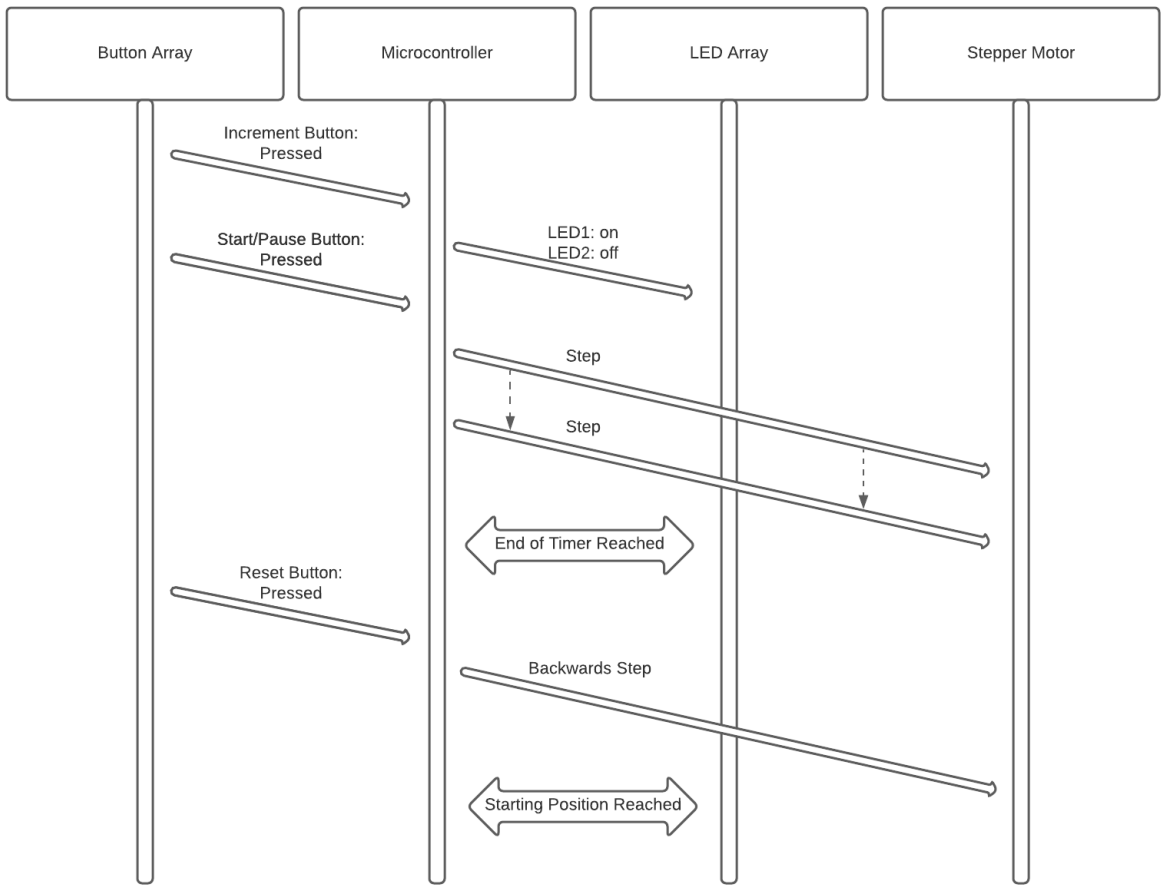
2. Requirements

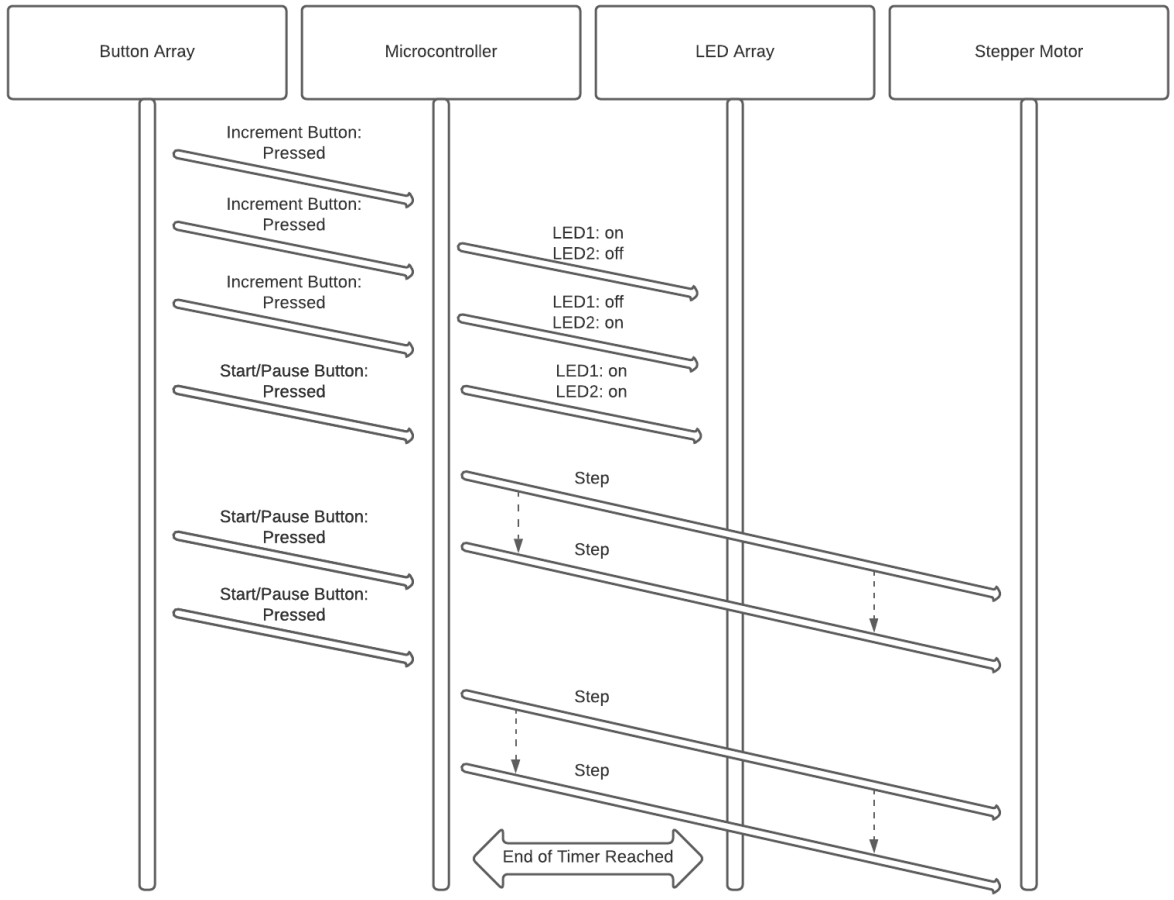
- R1: The default, starting position of the magnetic ball shall be the leftmost end of the track, in the STARTING mode.
- R2: The timer interval shall be initialized to 1 minute.
- R3: If the ball is in the starting position, i.e. in the STARTING mode, and the start/pause button is pressed, the ball should start moving across the track at a speed such that it reaches the end of the track in the timer interval.
    - R3a: If the ball is not in the starting position and not moving, i.e. in the PAUSED mode, and the start/pause button is pressed, we enter the RUNNING mode and the ball should start moving at the same speed as before.
- R4: If the ball is moving, in the RUNNING mode, and the start/pause button is pressed, we enter the PAUSED mode, and the ball should stop moving within 1 second.
- R5: There shall be two LEDs that visually display the timer interval.
    - R5a: If the timer interval is 1 minute, both LEDs shall be off.
    - R5b: If the timer interval is 2 minutes, LED1 shall be on and LED2 shall be off.
    - R5c: If the timer interval is 3 minutes, LED1 shall be off and LED2 shall be on.
    - R5d: If the timer interval is 4 minutes, both LEDs shall be on.
- R6: If the increment button is pressed and the timer interval is not 4 minutes and the ball is in the starting position, in the STARTING mode, the timer duration should be incremented by 1 minute and the LEDs should change to indicate the current timer interval.
    - R6a: If the increment button is pressed and the ball is not in the starting position or the timer interval is 4 minutes, there should be no change.
- R7: If the decrement button is pressed and the timer interval is not 1 minute and the ball is in the starting position, in the STARTING mode, the timer duration should be decremented by 1 minute and the LEDs should change to indicate the current timer interval.
    - R7a: If the decrement button is pressed and the ball is not in the starting position or the timer interval is 1 minute, there should be no change.
- R8: If the ball is not moving, i.e. is in the PAUSED mode or the FINISHED mode, and the reset button is pressed, the ball shall travel back to the starting position. The LEDs and timer duration shall remain the same.
    - R8a: If the ball is moving, i.e. in RUNNING mode, and the reset button is pressed, there should be no change to the system.
- R9: If the ball is at the end position at the rightmost of the track, in the FINISHED mode, pressing any of the buttons other than the reset button shall not change the system.
- R10: If the ball is moving, i.e. it is in the RUNNING mode and it has reached the end of the track, it shall stop moving and enter the FINISHED mode. The LEDs and timer duration shall remain the same.

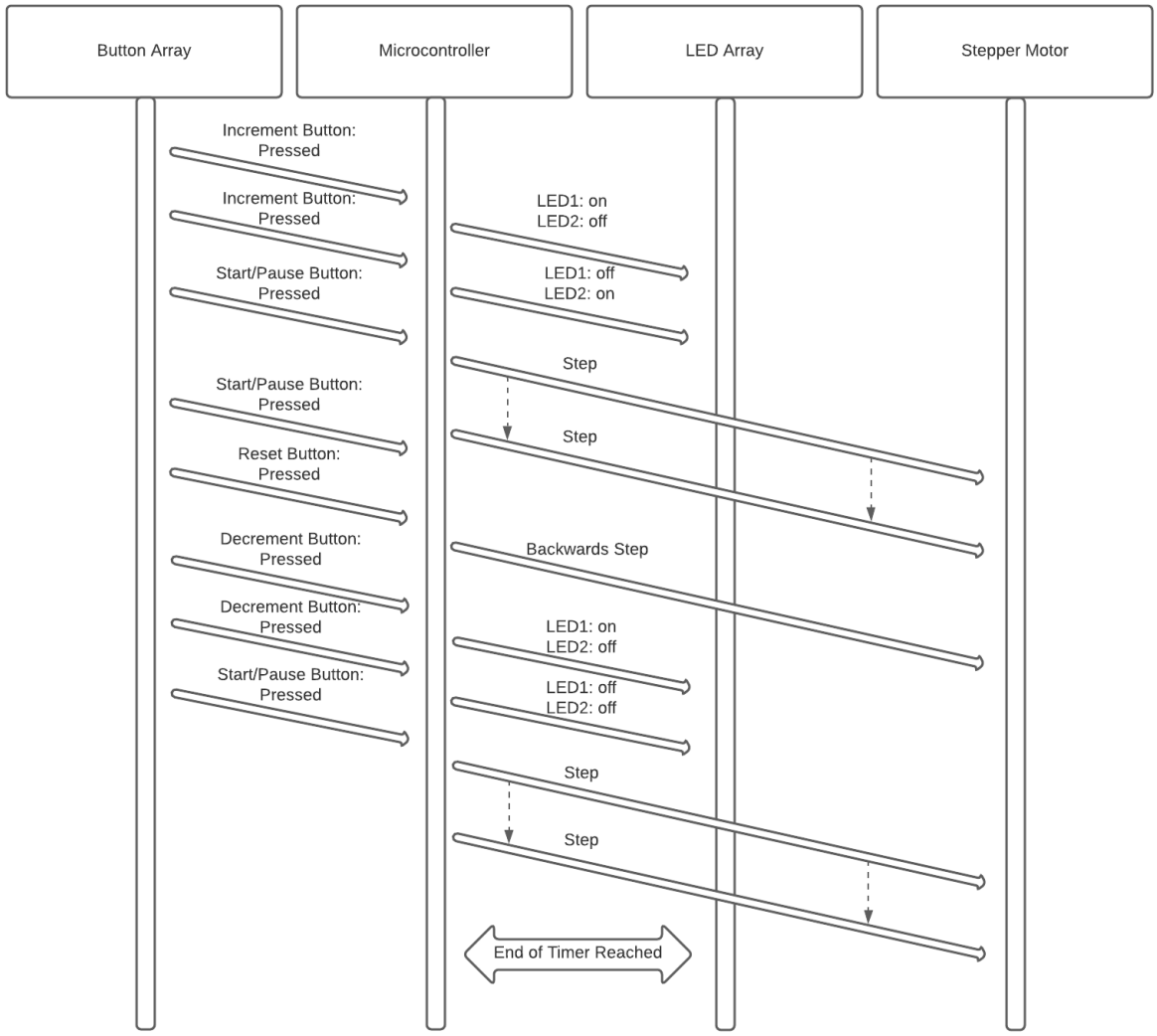
3. Architecture Diagram



4. Sequence Diagrams

Use case	Sequence Diagram
The user starts a 2-minute timer and allows it to run to the end, and then resets the timer.	

The user starts a 4-minute timer, pauses it, then resumes it again and allows it to run to the end.	

The user starts a 3-minute timer, pauses it, then resets it. Then, the user starts a 1-minute timer and allows it to run to the end.	

5. Finite State Machine

Constants:

Name	Description
`TRACK_DIST`	Total number of steps needed for the stepper motor to move the ball from one end of the track to the other

Inputs:

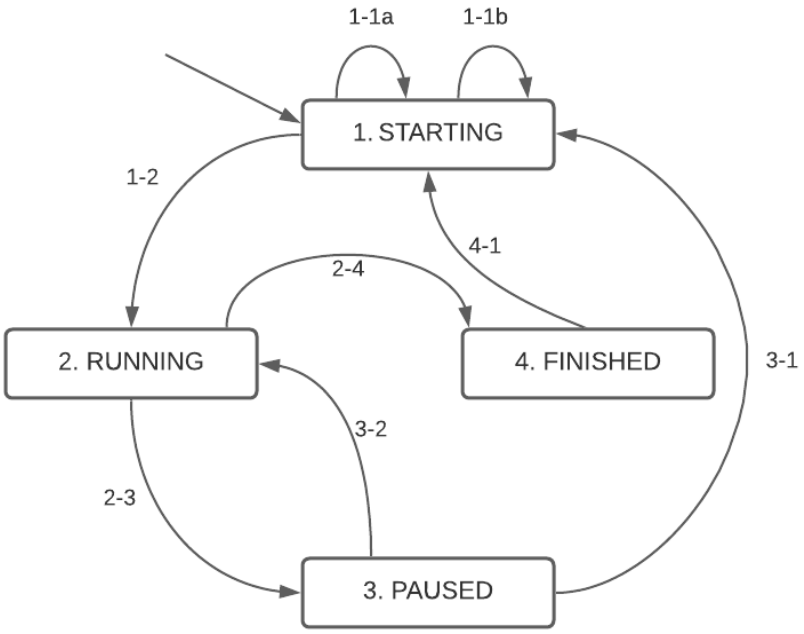
Name	Description	Initial value
`button_array [INC_BUTTON]`	Denotes whether the increment button has been pressed. 1 if pressed, 0 otherwise.	0
`button_array [DEC_BUTTON]`	Denotes whether the decrement button has been pressed. 1 if pressed, 0 otherwise.	0
`button_array [START_PAUSE_BUTTON]`	Denotes whether the start/pause button has been pressed. 1 if pressed, 0 otherwise.	0
`button_array [RESET_BUTTON]`	Denotes whether the reset button has been pressed. 1 if pressed, 0 otherwise.	0
`steps_taken`	Denotes how many steps the ball has taken.	0

Variables:

Name	Explanation	Initial value
`timer`	Denotes the amount of time in minutes the ball should take to travel from the start of the track to the end, given that it is not paused in its journey.	1
`freq_step`	Frequency to rotate the motor by in the `start_step` function, calculated using the track distance constant and the timer variable	`TRACK_DIST/ timer`

Output functions:

Name	Description
`set_lights(int timer)`	The function takes in the timer duration (which can have a value of 1, 2, 3, or 4) and sets (to HIGH or LOW) the two LEDs that represent the timer. If timer = 1, both LEDs are off. If timer = 2, LED1 is on and LED2 is off. If timer = 3, LED1 is off and LED2 is on. If timer = 4, both LEDs are on.
`start_step(int freq_step)`	Sets the correct timer/counter value and enables the timer/counter so that the interrupt moves the stepper motor at the desired frequency.
`stop_step()`	Disables the timer/counter.
`reset_system()`	Moves the ball back to its starting position on the track. Resets steps_taken to 0.



Transition	Guard	Output	Variables
1-1(a)	button_array[INC_BUTTON] = 1 ∧ timer < 4 ∧ button_array[DEC_BUTTON] = 0 ∧ button_array[START_PAUSE_BUTTON] = 0	set_lights(timer + 1)	timer := timer + 1 freq_step := TRACK_DIST/(timer*60)
1-1(b)	button_array[DEC_BUTTON] = 1 ∧ timer > 1 ∧ button_array[INC_BUTTON] = 0 ∧ button_array[START_PAUSE_BUTTON] = 0	set_lights(timer - 1)	timer := timer - 1 freq_step := TRACK_DIST/(timer*60)
1-2	button_array[START_PAUSE_BUTTON] = 1	start_step(freq_step)	
2-3	button_array[START_PAUSE_BUTTON] = 1 ∧ ¬(steps_taken = TRACK_DIST)	stop_step()	
2-4	steps_taken = TRACK_DIST	stop_step()	
3-1	button_array[RESET_BUTTON] = 1	reset_system()	
3-2	button_array[START_PAUSE_BUTTON] = 1 ∧ button_array[RESET_BUTTON] = 0	start_step(freq)	
4-1	button_array[RESET_BUTTON] = 1	reset_system()	

The FSM has been fixed based on the feedback provided in the peer review, which can be found in Appendix A - Peer Review.

6. Traceability Matrix

	R1	R2	R3	R3a	R4	R5	R5a	R5b	R5c	R5d	R6	R6a	R7	R7a	R8	R8a	R9	R10
STATE																		
1	x	x				x	x	x	x	x		x		x				
2																x		
3					x													
4																	x	x
TRANS ITION																		
1-1(a)											x							
1-1(b)													x					
1-2			x															
2-3					x													
2-4																		x
3-1															x			
3-2				x														
4-1															x		x	

7. Testing

Unit Testing

We started by unit testing the button array, LED array, stepper motor, and FSM separately. The individual components were unit tested with the following functions:

Component	Test Function	Description
Button array	**`test_button_handlers()`**	Prompts the user to press the buttons in a specific order, prints out to the Serial Monitor to show that the interrupt has been triggered, and checks that the flags are set correctly. This unit test is sufficient as all the buttons and their corresponding interrupt service routines are tested.
LED array	**`test_leds()`**	Calls the `set_light` functions with the timer values 1, 2, 3, and 4 in order, allowing the user to visually inspect the correct LEDs lighting up. This unit test is sufficient as all allowable inputs into the `set_light` function which sets the two LEDs are tested.
Stepper motor	**`test_stepper()`**	Rotates the stepper motor forwards and backwards, allowing the user to visually inspect the correct movement of the stepper motor. This ensures that all four pins of the stepper motor have been connected correctly. This unit test is sufficient as both the clockwise and anticlockwise movement of the stepper motor are tested.
FSM	**`test_fsm()`**	Checks the `update_fsm` function to ensure that (a) the correct state is returned, (b) the variables are updated correctly, and (c) the output functions are called with the correct arguments, given an initial state, inputs, and initial variables. In this function, we performed white-box testing and achieved 100% branch and line coverage to ensure that all the transitions are correct. We visually confirmed that we met this criteria. The specific tests can be found in Appendix B - FSM Tests. Full transition coverage was also achieved as our tests check not only for transitions to different states, but also default transitions where we remain in the same state.

In general, our unit tests achieve good coverage of all possible inputs, as our use of buttons as the only inputs means that there are a limited number of possible inputs into the system.

Integration Testing

We then moved on to integration testing to test the interfaces between the components. We tested the following interfaces:

Components	Tests performed
Button array, LED array, FSM	● Both LEDs are off when the program is first executed. ● If the stepper motor is rotating (sRUNNING) or is not at the initial position (sPAUSED, sFINISHED), clicking either the increment or decrement button does not change the LEDs. **Increment button** ● If both LEDs are off, clicking the increment button causes LED1 to turn on. ● If LED1 is on and LED2 is off, clicking the increment button causes LED1 to turn off and LED2 to turn on. ● If LED1 is off and LED2 is on, clicking the increment button causes both LEDs to turn on. ● If both LEDs are on, clicking the increment button does not change the LEDs. **Decrement button** ● If both LEDs are off, clicking the decrement button does not change the LEDs. ● If LED1 is on and LED2 is off, clicking the decrement button causes both LEDs to turn off. ● If LED1 is off and LED2 is on, clicking the decrement button causes LED1 to turn on and LED2 to turn off. ● If both LEDs are on, clicking the decrement button causes LED1 to turn off and LED2 to turn on.
Button array, stepper motor, FSM	● If the stepper motor is not moving and not in the final position (sSTARTING, sPAUSED), clicking the start/pause button causes the stepper motor to turn clockwise steadily. ● If the stepper motor is moving (sRUNNING), clicking the start/pause button causes the stepper motor to stop moving. ● If the stepper motor is not not moving and not in its initial position (sPAUSED, sFINISHED), clicking the reset button causes the stepper motor to rotate anticlockwise back to the initial position. ● If the stepper motor is moving or in the initial state (sRUNNING, sSTARTING), clicking the reset button does not affect the movement of the stepper motor.

Our integration testing is sufficient as all the possible interfaces between the components were tested. To ensure that the tests pass consistently, we also ran each test thrice.

System testing

Lastly, we used the specified requirements and the use cases in our sequence diagrams to perform system testing.

To perform software testing, we printed to the Serial Monitor to ensure that the correct state transitions occur and the output functions are called with the correct arguments based on the buttons pressed.

To perform user-facing/acceptance testing, we also checked that the LED lights are set correctly and the stepper motor moves in the correct direction or stops moving correctly based on the buttons pressed or the time passed. Note that our acceptance test involves only the bare stepper motor as we do not have a working conveyor belt system.

Once again, we tested each requirement thrice to ensure consistency.

Based on our system testing, we were in fact able to identify an issue with our overall system. We noticed, from both software and acceptance testing, that our timer duration was not accurate, and initially gave us the following errors for the different durations:

Duration of Timer (min)	Mean error (s)
1	+2
2	+4
3	+11
4	+8

We came up with several theories:

- New timer/counter interrupts were being triggered before earlier ones were serviced. In this case, however, the error would decrease as the duration of the timer increases and the frequency of interrupts decreases.
- The timer/counter interrupt handler was taking too long to run. We minimized our timer/counter interrupt handler code, but this had no effect on the error.
- We did not think that it was due to interference by other interrupts, as none of the buttons were pressed when we were testing the timer/counter.

In the end, we tried changing the timer/counter clock from the Ultra Low-Power Internal Oscillator (OSCULP32K) to the Crystal Oscillator (XOSC32K). This removed the error for the timer durations of 1, 2, and 4 min, but not for 3 min.

Duration of Timer (min)	Mean error (s)
1	0
2	0
3	+5
4	0

Upon further inspection, we realized that the error for the 3-minute timer was due to the fact that the frequency required for the 3-minute timer to work properly was not a whole number, 13.3333333 Hz. This resulted in the clock frequency being divided by a smaller, rounded-down number when calculating the number of cycles. The larger number of cycles led to the total time taken overshooting by 5 seconds.

The difficulty we faced in testing and debugging our system attests to the fact that our heavy reliance on a timer/counter complicates the testing process.

8. Safety and Liveness Requirements

Safety requirements:

- The value of the timer duration in minutes is always 1, 2, 3, or 4.
    - G (timer = 1 ∨ timer = 2 ∨ timer = 3 ∨ timer = 4)
- The distance traveled by the ball magnet is never greater than the distance of the track.
    - G (¬(steps_taken > TRACK_DIST))

Liveness requirements:

- The electrical signal sent to the LED lights matches the value of the timer duration (both LOW if 1, LED1 HIGH/LED2 LOW if 2, LED1 LOW/LED2 HIGH if 3, both HIGH if 4).
    - G (timer = 1 ⇒ F (LED 1 = LOW ∧ LED 2 = LOW)) ∧
      (timer = 2 ⇒ F (LED 1 = HIGH ∧ LED 2 = LOW)) ∧
      (timer = 3 ⇒ F (LED 1 = LOW ∧ LED 2 = HIGH)) ∧
      (timer = 4 ⇒ F (LED 1 = HIGH ∧ LED 2 = HIGH)))
- If the system is in the PAUSED state, and the reset button is pressed, the ball magnet will return to the initial position and the system enters the STARTING state.
    - G (PAUSED ∧ button_array[RESET_BUTTON] ⇒
      F (STARTING ∧ steps_taken = 0))
- If the system is in the STARTING state and the start/pause button is pressed, the system enters the RUNNING state.
    - G (STARTING ∧ button_array[START_PAUSE_BUTTON] ⇒ F (RUNNING))

9. Modelling

The environmental processes that we need to model in our system to create a close system for analysis are the clicking of the buttons by a user, the triggering of the timer/counter interrupts to increment `steps_taken`, and the passage of actual time.

Since the user can click any of the four buttons (start/pause, reset, increment, decrement) at any point of time and the clicking of the button generates a pure signal (i.e., present or absent), the clicking of each of these buttons should be modelled as a discrete, non-deterministic system.

Our timer/counter triggers an interrupt at a fixed frequency based on the duration of the timer set by the user, and the interrupt handler increments the `steps_taken` variable by 1. The design of our FSM means that the `steps_taken` variable can only take on an integer value in [0, TRACK_DIST]. As such, the trigger of the interrupt to increment `steps_taken` should be modelled as a discrete, deterministic system.

To analyse the correctness of our system, we will need to keep track of the actual time that has elapsed between the start of the timer and the end of the timer. We will therefore need to model time as a continuous, deterministic system with a derivative of 1, meaning that the system simply trivially outputs the actual time based on a particular point of reference. We can then use the calculations to ensure that the computed elapsed time equals the actual elapsed time.

10. Code Deliverable

- Files
  - `timer.h` - Defines constants, declares global variables, and declares function headers.
  - `timer_utils.ino` - Defines output functions used by the FSM. Defines interrupt handlers used to get button inputs. Defines interrupt handlers used for the timer/counter and watchdog timer.
  - `timer.ino` - The main file that contains the `setup` and `loop` functions. Defines the `update_fsm` function called in the main loop that implements the functionality of the timer.
  - `timer_tests.ino` - Contains functions to perform unit tests on the FSM, button array, LED array, and stepper motor.
- Requirements
  - PWM - controlling stepper motor
    - `timer_utils.ino`: 125 (`TC3_Handler`), 87 (`reset_system`)
  - Watchdog timer
    - `timer.ino`: 60-81 (`setup`), 95 (`loop`)
    - `timer_utils.ino`: 129-134 (`WDT_Handler`)
  - Interrupt Service Routine
    - `timer.ino`: 30-35 (`setup`)
    - `timer_utils.ino`: 12-38 (button handlers),
  - Timer/Counter
    - `timer.ino`: 37-58 (`setup`)
    - `timer_utils.ino`: 60-78 (`start_step`, `stop_step`)
    - The calculations to determine the clock frequency are shown in Appendix C - Timer/Counter Calculations

11. Running Unit Tests

For all tests, first:

1. Comment out the body of `loop` in `timer.ino`
2. Comment out the WDT code from `setup` in `timer.ino`

To test the button array:

1. Uncomment `#define TESTING` at the top of `timer.ino`
2. Uncomment the call to `test_button_handlers` at the bottom of `setup` in `timer.ino`
3. Make sure the circuit is connected properly (see Appendix D - Circuit Diagram)
4. Run the code
5. Open the Serial Monitor
6. Press the buttons and check that the appropriate messages are printed on the Serial Monitor

To test the LED array:

1. Uncomment the call to `test_leds` at the bottom of `setup` in `timer.ino`
2. Make sure the circuit is connected properly (see Appendix D - Circuit Diagram)
3. Run the code
4. Open the Serial Monitor
5. Observe the LEDs to make sure they are lighting up as expected

To test the stepper motor:

1. Uncomment the call to `test_stepper` at the bottom of `setup` in `timer.ino`
2. Make sure the circuit is connected properly (see Appendix D - Circuit Diagram)
3. Run the code
4. Watch the stepper motor to make sure it turns both clockwise and counterclockwise

To test the FSM:

1. Uncomment `#define TESTING` at the top of `timer.ino`
    a. The mocked versions of the output functions `set_lights`, `start_step`, `stop_step`, and `reset_system` are used if the macro is defined. These functions do not actually interact with the stepper motor or LED lights, but instead set the mocked variables `mocked_timer` and `mocked_freq_step` to allow the checking of the arguments into the output functions.
2. Uncomment the call to `test_fsm` at the bottom of `setup` in `timer.ino`
3. Run the code
4. Open the Serial Monitor
5. Check it says "All FSM Tests Passed!"

12. Reflections

We were unable to meet our desired goals due to hardware issues which are explained below.

The challenges we encountered had to do with reasoning about the FSM, using the stepper motor library with interrupts, and having the physical components work together correctly.

Our initial reasoning about the environment, inputs, and variables of the FSM was faulty. We had trouble understanding how and where to reset the flags that tell us whether a button has been clicked or not. Initially, we had them as variables that our FSM reset but it is incorrect to have them as variables as they are influenced by the environment. Hence, we made them a part of the environment and passed them into the FSM as inputs. The environment itself resets the flags.

Using the `Stepper` library within the timer/counter interrupt handler created problems as the `step` function was blocking. We alleviated this problem by using the `AccelStepper` library instead, which provides non-blocking functions to control the stepper motor.

The main problem with our project had to do with the hardware/physical components. Our goal was to use the stepper motor to turn a gear that would move a conveyor belt with a ball magnet attached to it. However, the stepper motor that we currently have cannot produce sufficient torque to move them. In addition, our conveyor belt also does not have sufficient friction to firmly attach to the gear and tension to support the weight of the ball magnet. This means that when we put the physical parts together, we are unable to move the ball magnet as desired for the timer to function properly. If we had more time and better resources (e.g. stronger stepper motor, linear rail, appropriate conveyor belt), we could iterate through the physics of our system to produce a product that meets our goals. The lack of torque from the stepper motor also meant it had to work really hard and consumed a lot of power from the batteries very quickly.

Appendix A - Peer Review

Project name:	Robotic Timer	
Peer reviewer roles		
Manager:	Ian Rackow	
Reader:	Jason Crowley	
Recorder:	David Hong	
Defect record		
Transition or state #	**Defect**	**Remarks**
N/A	We don't have inputs, outputs, and variables explicitly defined; they're initialized and intuitive, but they're not described.	Fixed
1-1a	Not mutually exclusive with 1-1b OR 1-2	Fixed
1-1b	"	Fixed
1-2	"	Fixed
2-3	Not mutually exclusive with 2-4	Fixed
2-4	"	Fixed
3-1	Not mutually exclusive with 3-2	Fixed
3-2	"	Fixed
1-1a and 1-1b	Misaligned FSM and intended behavior/requirements: Suppose the inc_button is pressed while the timer is 4. Then, suppose the dec_button is pressed. The intended behavior is that 1-1b triggers and the timer falls to 3. The actual behavior is that 1-1b will immediately waterfall into 1-1a, so the timer will fall to 3 but instantly rise again to 4.	Fixed

Peer reviewer roles		
Manager:	David Gralla, Morgann Thain	
Reader:	Floria Tsui	
Recorder:	Monica Roy	
Defect record		**Remarks**
Transition or state #	**Defect**	
State start	no defect	
State numbers	no defect	
State names	no defect	
Inputs, outputs, variables defined	outputs are not defined but do seem somewhat straightforward, also a bit hard to tell which are inputs and which are variables	Fixed
Initializations	no defect	
Transitions	Maybe not necessary to have both 1-1(a) and 1-1(b), also maybe put the parenthesis so it doesn't look like 1a and 1b are states, transitions are mutually exclusive when there are multiple out of one state	Fixed
Inputs and variables checked in guard	no defect	
Outputs and variables set in actions	no defect, but again a bit hard to tell if some of those are variables or inputs	Fixed
FSM behavior	for start_pause, it would make sense to create a separate function that detects the rising and falling edges and call that function in your FSM instead of directly worrying about that in the FSM itself (idea from Milda)	Fixed

Appendix B - FSM Tests

Tests			INPUTS					VARIABLES		OUTPUT	
Test	Start state	End state	button_array[START_PAUSE_BUTTON]	button_array[RESET_BUTTON]	button_array[INC_BUTTON]	button_array[DEC_BUTTON]	steps_taken	timer	freq_step	function	argument list
0	1	1	0	0	1	0	0	2 3	20 13.33333333	set_lights	3
1	1	1	0	0	0	1	0	2 1	20 40	set_lights	1
2	1	1	0	0	1	0	0	4 4	10 10	-	-
3	1	1	0	0	0	1	0	1 1	40 40	-	-
4	1	1	0	0	1	1	0	2 2	20 20	-	-
5	1	2	1	0	0	0	0	3 3	13.33333333 13.33333333	start_step	13.33333333
6	2	3	1	0	0	0	40	4 4	10 10	stop_step	-
7	2	4	0	0	0	0	2400	2 2	20 20	stop_step	-
8	2	2	0	0	0	0	40	2 2	20 20	-	-
9	3	1	0	1	0	0	40	1 1	40 40	reset_system	-
10	3	2	1	0	0	0	40	3 3	13.33333333 13.33333333	start_step	13.33333333
11	3	3	0	0	0	0	40	3 3	13.33333333 13.33333333	-	-
12	4	1	0	1	0	0	2400	4 4	10 10	reset_system	-
13	4	4	0	0	0	0	2400	4 4	10 10	-	-

Appendix C - Timer/Counter Calculations

Timer duration	Note frequency	Toggle every _ cycles	Column B Rounded Down	Actual frequency	% Error	Actual Time
1	40	408.475	408	40.04656863	0.1164215686	59.93022829
2	20	816.95	816	20.02328431	0.1164215686	119.8604566
3	13.33333333	1225.425	1225	13.33795918	0.03469387755	179.9375727
4	10	1633.9	1633	10.00551133	0.05511328843	239.867801
		CLOCK_FREQ:	32678			
		TRACK_DIST:	2400			

Appendix D - Circuit Diagram