Twiddle Lock Project

EE3096S

Jonathan Oehley – OHLJON001

Ben Adey – ADYBEN001

November 2018

Table of Contents

Plagiarism Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and to pretend that it is one's own.
2. I have used the Harvard convention for citation and referencing. Each significant contribution to, and quotation in, this essay from the work, or works, of other people has been acknowledged through citation and reference.
3. This work-piece is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

[illegible]

Signature

1 June 2018

Date

[illegible]

Signature

Introduction

This report details the design, construction, and testing of an electronic form of a Dial Combination Safe (DCS) which will hence forth be referred to as the “Twiddle Lock”.

The Twiddle Lock consists, in its most important basic components, of a Raspberry Pi Model 3B, and ADC and a potentiometer. The user will turn the potentiometer in order to enter a code to unlock the ‘Twiddle Lock’. A pushbutton is used to activate the sensing of a code input.

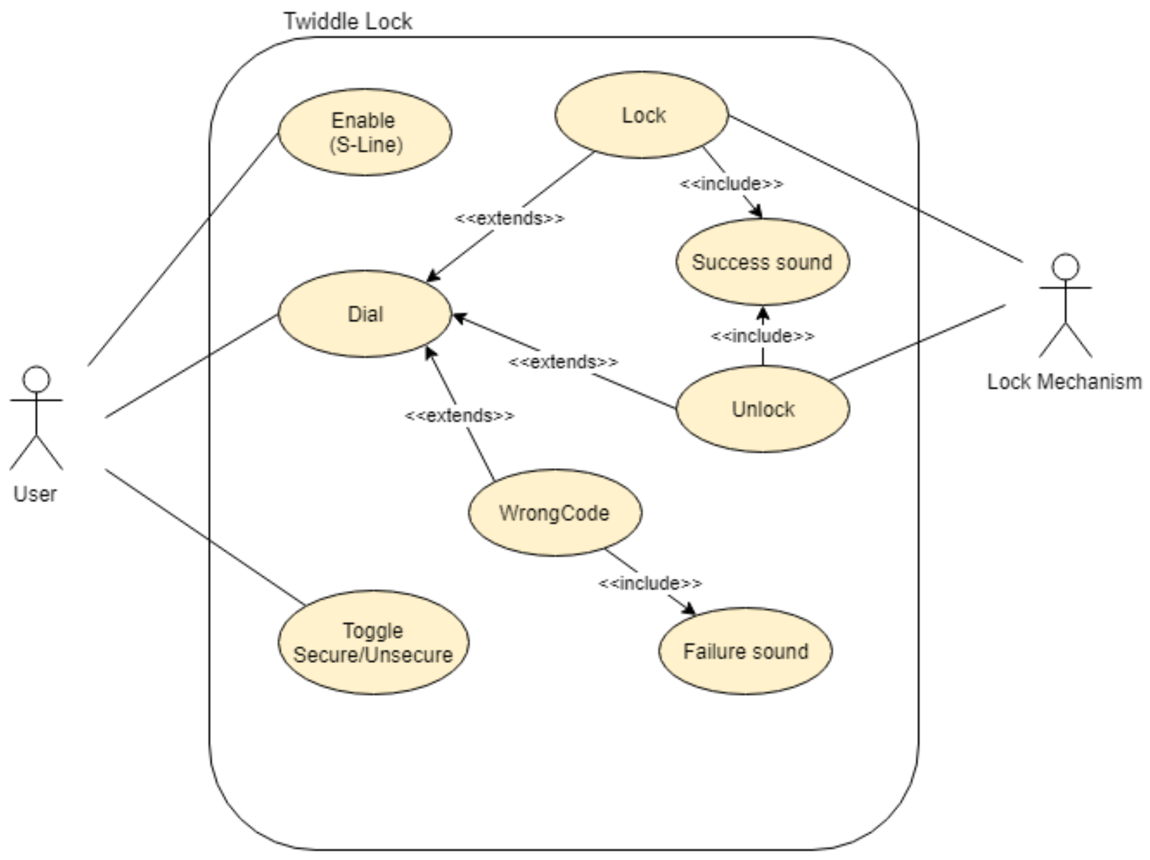
The design also included three output LEDS which represented the states of the lock itself, the lock signal line and the unlock signal line.

Additionally, a speaker was connected to the Raspberry Pi in order to play sounds upon locking, unlocking and failing to open the lock

This report first details the operational requirements of the Twiddle Lock through an UML Use Case Chart. It then goes on to describe the specifications and design of the implementation of the Twiddle Lock. The operation of the lower level code is then shown in the implementation section. The results from the validation steps are thereafter provided and the performance of the system analysed. Finally, the report is concluded.

Requirements

Figure 1: UML Use Case Diagram



The above diagram shows the operation of the Twiddle Lock from the user's perspective. Each use case has an observable effect on the system, noticeable by the user.

The only change that was made to the operation of the Twiddle Lock is that when a combocode is specified as the lock/unlock code for the lock, the number represents a number of seconds instead of hundreds of milliseconds as described in the project brief. Additionally, the tolerance was increased to 500ms for ease of use

Specification and Design

Figure 2: UML State Chart

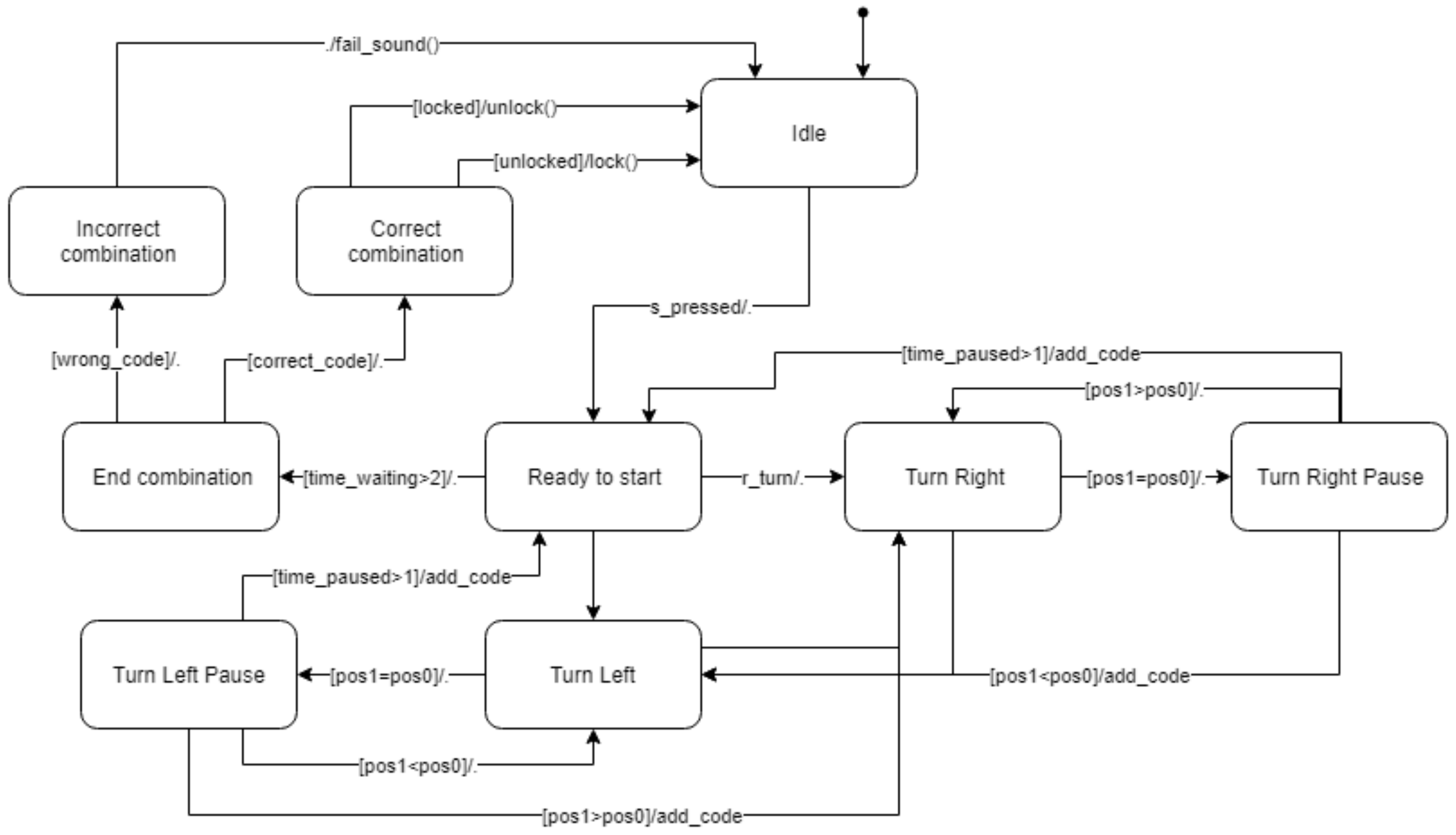


Figure 2 shows the detailed state follow and control of the program that governs the operation of the Twiddle Lock. The state control is governed by the mode variable in code. The pseudocode events, conditions and outputs are written in part for readability (and don't have exact equivalents in code) and in part refer to variables in code (such as pos0, and pos1 which refer to the previous and current ADC readings). In all of the cases, however, there diagram and code are logically equivalent.

Figure 3: UML Deployment Diagram

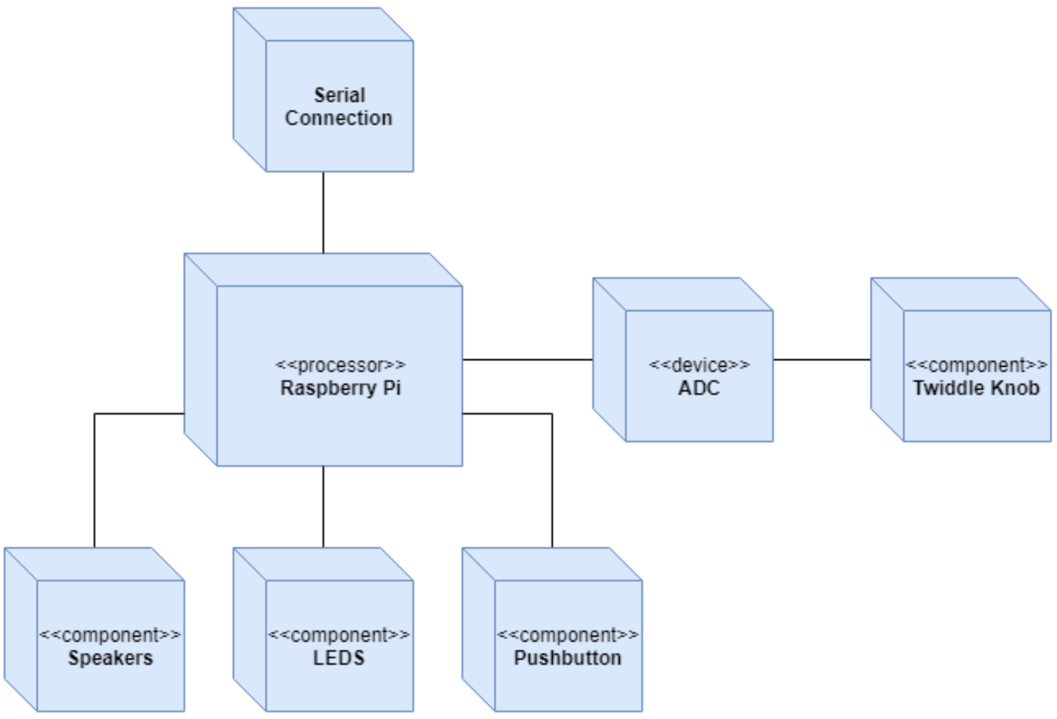


Figure 3 shows the model of the embedded system as the combination of the various hardware nodes. Each component that was use is represented as a node in the system. Additionally, the serial connection to a laptop connection was represented a software node.

Implementation

The following section explains some important sections of code in the program. The code sections discussed here are: the main while loop, the ADC reading function, the combination check function.

Main while loop

Each state in the state chart is represented by a *mode* in the main while loop of the program. Each of the modes is coded as an if statement. An example is the turn right mode, which is shown below:

```
elif mode == 2:  # turn_right mode
                if pos_1 == pos_0:
                    # Record time, position, set mode to turn_right_pause:
                    pos_0 = pos_1
                    t_pause_start = time.time()
                    mode = 4    # mode -> turn_right_pause
                elif pos_1 < pos_0:
                    # Direction has changed. Record code and start new code
                    t_turn_stop = time.time()
                    if (t_turn_stop - t_turn_start > 0.2):
                        add_code(t_turn_start, t_turn_stop, 1)

                        t_turn_start = time.time()
                        pos_0 = pos_1
                        mode = 3 # mode -> turn_left
                    else:
                        pos_0 = pos_1
                else:
                    pos_0 = pos_1
```

In *turn right mode*, there are three possibilities:

1. **pos_1 == pos_0**: In this case, the time and position are recorded and the mode is set to *pause mode*
2. **pos_1 < pos_0**: If the new position is less than the previous position, the twiddle knob has changed direction.
3. **pos_1 > pos_0**: This case means that the twiddle knob has continued to turn right, and no changes need to be made. The program continues to track time and position of the knob.

The program is always in one of the following modes, which each behave similarly to *turn right mode*. Each mode corresponds to a value of the integer *mode*:

- Idle (mode = 0)
- Ready to start (mode = 1)
- Turn right (mode = 2)
- Turn left (mode = 3)
- Turn right pause (mode = 4)
- Turn left pause (mode = 5)
- End combination (mode = 10)

Exponential smoothing of ADC reading

The *read_adc()* function reads the ADC to determine the position of the twiddle knob. The code used is shown below:

```
def read_adc():
    global numReadings
    global readIndex
    global total
    global average
    global read_avg
    global readings

    # subtract the last reading:
    total = total - readings[readIndex]
    # read from the sensor:
    readings[readIndex] = mcp.read_adc(0)
    # add the reading to the total:
    total = total + readings[readIndex]
    # advance to the next position in the array:
    readIndex = readIndex + 1

    # if we're at the end of the array...
    if (readIndex >= numReadings):
        #...wrap around to the beginning:
        readIndex = 0

    # calculate the average and round reading:
    average = total / numReadings
    return int( (average/1023) * 127)
```

** Code implementation based on Arduino code tutorial [2].*

The code to read from the ADC uses exponential smoothing [1]. Without the smoothing, the ADC value can jump between values, even when the twiddle knob (potentiometer) is stationary. The exponential smoothing works by keeping a simple moving average of past readings [2].

Check Combination code

The *check_combination* function is used to check the combination entered by the user.

First, two arrays are created as follows to represent the correct combination of the combination lock:

```
def check_combination(code, durations, directions, tolerance):
    code_durations = []
    code_directions = []
```

The arrays are populated as shown below. The string *code* is the pre-determined combination hard-coded as a constant into the program. In this program, *code* has the value **L1R2L3.**

```
    for i in range (int(len(code)/2)):
        if code[2*i] == 'L':
            code_directions.append(0)
        else:
            code_directions.append(1)

        code_durations.append(int(code[2*i+1]))
```

Next, the function checks whether the code entered by the user matches the pre-set code. This is done by comparing each value in the combinations and durations arrays. The check includes a tolerance value. This tolerance value is a constant set at the start of the program. If a user-entered duration value is within the tolerance of the pre-set duration value, it is considered a match.

```python
    for i in range (len(code_durations)):
        if durations[i]*10 < (code_durations[i]*10 -  tolerance/100) or durations[i]*10 > (code_durations[i]*10 +  tolerance/100):
            return False
        if code_directions[i] == directions[i]:
            continue
        else:
            return False
    return True
```

If the code entered is correct, the function returns True, otherwise False.

Check combination in unsecure mode

The following code checks the code entered by the user in unsecure mode:

```python
def check_unsecure(code, durations, tolerance):
    code_durations = []

    for i in range (int(len(code)/2)):
        code_durations.append(int(code[2*i+1]))

    code_durations = sort(code_durations)
    durations = sort(durations)

    for i in range (len(code_durations)):
        if durations[i]*10 < (code_durations[i]*10 -  tolerance/100) or durations[i]*10 > (code_durations[i]*10 +  tolerance/100):
            return False

    return True
```

This code works similarly to the code use to check combinations in secure mode. The difference is that in unsecure mode, only the durations need to match, and order is unimportant. Thus, so that order can be ignored, a sort function is used to sort both durations arrays into order of increasing time values.

Validation and Performance

Test Cases

Tests were conducted to compare the manual timing of a combination code to what the computer reads.

In each test, a smartphone stop watch was used to manually time the entry of the combination code. The results are shown in the table below. For the purposes of testing, times recorded by the Twiddle Lock system are rounded to 2 decimal places. Times are given in seconds.

Test case		Times (direction alternates between L and R)									Average Deviation (s)
1	System	2.02	2.88	2.8	2.91	2.88	2.52	2.86	2.34	21.21	0.17875
	Manual	2.06	3.03	2.91	2.99	3.00	2.83	2.98	2.84	22.64	
2	System	1.21	1.09	1.21	1.11	1.27	1.36	1.47	1.25	9.97	0.05125
	Manual	1.14	1.23	1.31	1.17	1.36	1.36	1.51	1.3	10.38	
3	System	0.86	0.77	0.95	0.89	1.11	0.94	0.95	0.9	7.37	0.08
	Manual	0.98	0.91	1.04	0.96	1.05	1	1.01	1.06	8.01	

As can be seen from the test cases, the Twiddle Lock system times do not differ significantly from the manual timing. The manual timing was on average longer than the system timing. This can be attributed to the delay between each timing in the python program. The delay is caused by the program executing various functions between stopping the timer and starting again.

The system timing is adequate for the purposes of the twiddle lock. It should be noted that the accuracy of the manual timing is susceptible to human error.

Snapshots from each of the tests are shown in Appendix A.

Conclusion

The Twiddle Lock system was able to meet the design requirements specified at the start of this report. It was shown that a code could be entered by the user and be read as either correct or incorrect.

One potential problem in the system was that the timing was not always reliable. Otherwise, the system could be considered to be successful.

A system working in this way would be limited in its usefulness. There must be a tolerance in checking whether a code is correct since people cannot accurately and consistently enter the same time. Some people can reproduce time measurements more accurately than others. It may take multiple tries for users to enter the correct code.

In this implementation of the Twiddle Lock for example, a tolerance of 500 ms was used. By using this tolerance, the correct combination could be entered repeatedly. However, the trade-off is that the number of possible code combinations that can be entered within a reasonable amount of time is low compared to other locking systems.

This Twiddle Lock would be useful for people who can accurately and repeatedly enter time duration combinations. It would be a unique and novel way to unlock a safe and one which trained criminals would not be prepared for.

References

[1]"How to smooth potentiometer values", *Electrical Engineering Stack Exchange*, 2018. [Online]. Available: https://electronics.stackexchange.com/questions/64677/how-to-smooth-potentiometer-values. [Accessed: 06- Nov- 2018].

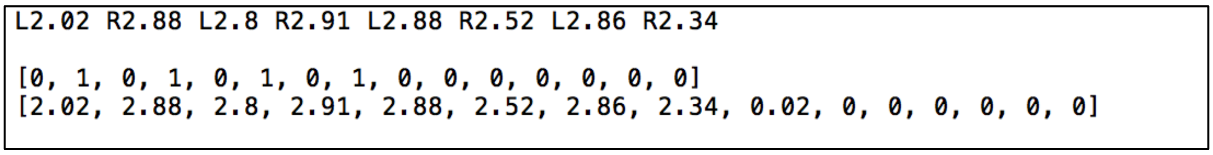
[2]"Arduino - Smoothing", *Arduino.cc*, 2018. [Online]. Available: http://Arduino. [Accessed: 06- Nov-2018].

[3]"Exponential smoothing", *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Exponential_smoothing. [Accessed: 06- Nov- 2018].

Appendix A – Validation Testing

Below are the snapshots for each of the tests

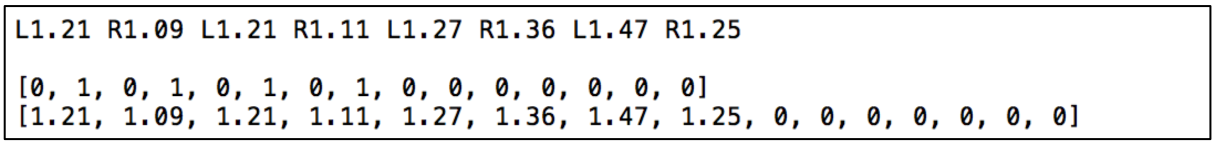
Test 1



```
L2.02 R2.88 L2.8 R2.91 L2.88 R2.52 L2.86 R2.34

[0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[2.02, 2.88, 2.8, 2.91, 2.88, 2.52, 2.86, 2.34, 0.02, 0, 0, 0, 0, 0, 0]
```

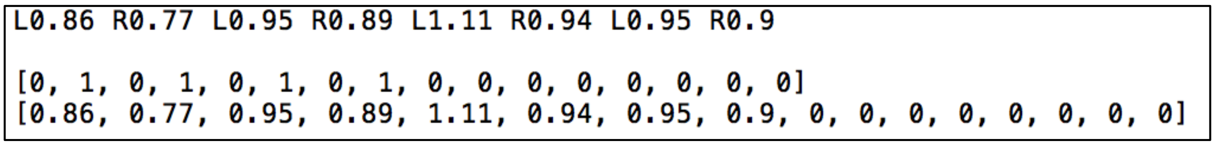
Test 2



```
L1.21 R1.09 L1.21 R1.11 L1.27 R1.36 L1.47 R1.25

[0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[1.21, 1.09, 1.21, 1.11, 1.27, 1.36, 1.47, 1.25, 0, 0, 0, 0, 0, 0, 0]
```

Test 3



```
L0.86 R0.77 L0.95 R0.89 L1.11 R0.94 L0.95 R0.9

[0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0.86, 0.77, 0.95, 0.89, 1.11, 0.94, 0.95, 0.9, 0, 0, 0, 0, 0, 0, 0, 0]
```