EEE3096S
Prac06
Project A: Twiddle Lock

OLVJAR002	KKRSNE001
Jarrod Olivier	Snehin Kukreja

Introduction:

A client has requested the design and implementation of a dial combination lock called the 'Twiddle Lock'.
The design will make use of Raspberry Pi Model 3B as the 'Lock Controller' making use of it's SOC and GPIO to monitor and control the lock as well as the onboard AUX port which will be connected to a speaker. For the Dial, to save on the cost of the prototype - a generic and cheap potentiometer will be used, this will be connected to an MCP3008 ADC, appropriate tolerances have been chosen. LED's and a speaker will be used to give the user feedback; i.e. if a correct or incorrect combination has been entered.
The Lock Controller uses a discrete time interval, for timing, to compare readings from the Dial channel of ADC.
The user can select between two modes of operation of the Twiddle Lock, secure mode and unsecure mode, by pressing one of two push buttons. In secure mode the combination refers to the sequence and respective duration of rotation of the dial; while the combination, in unsecure mode, refers to an ascending sorted array of durantions.
The user gave specific details on sorting the time array in unsecure mode as such an insertion sort in arm assembly has been developed and attached to the submission of this report.

Requirements:

The user requires a locking device which will be referred to as the 'Twiddle Lock' in this report, originally the user suggested that one push button would toggle the mode of operation (secure/unsecure) and another would initiate input monitoring of the rotating dial i.e. a service line. However this was changed such that one pushbutton would set secure mode and another would set unsecure mode with a timer beginning immediately after either button was pressed.
This change is beneficial for the user, as the user can press a labeled button of the mode of operation of choice and enter a combination; instead of previously not quite knowing what mode of operation the device is in.
Within the 'Twiddle Lock' there is a hardcoded sequence and duration stored in two separate arrays, lockTime[] for the duration of each turn and lock[] for the direction of the turn the dial, 0 (left) or 1(right).
As python uses dynamic arrays, the size of the array does not need to be declared. When the user selects one of the modes of operation (secure/unsecure) both input arrays, Time[] and code[], are emptied.
The user will receive feedback about the code they have entered by having a sound played and an LED turn on for 2[s].

The above is implemented in the use case model displayed in Figure 1 below:

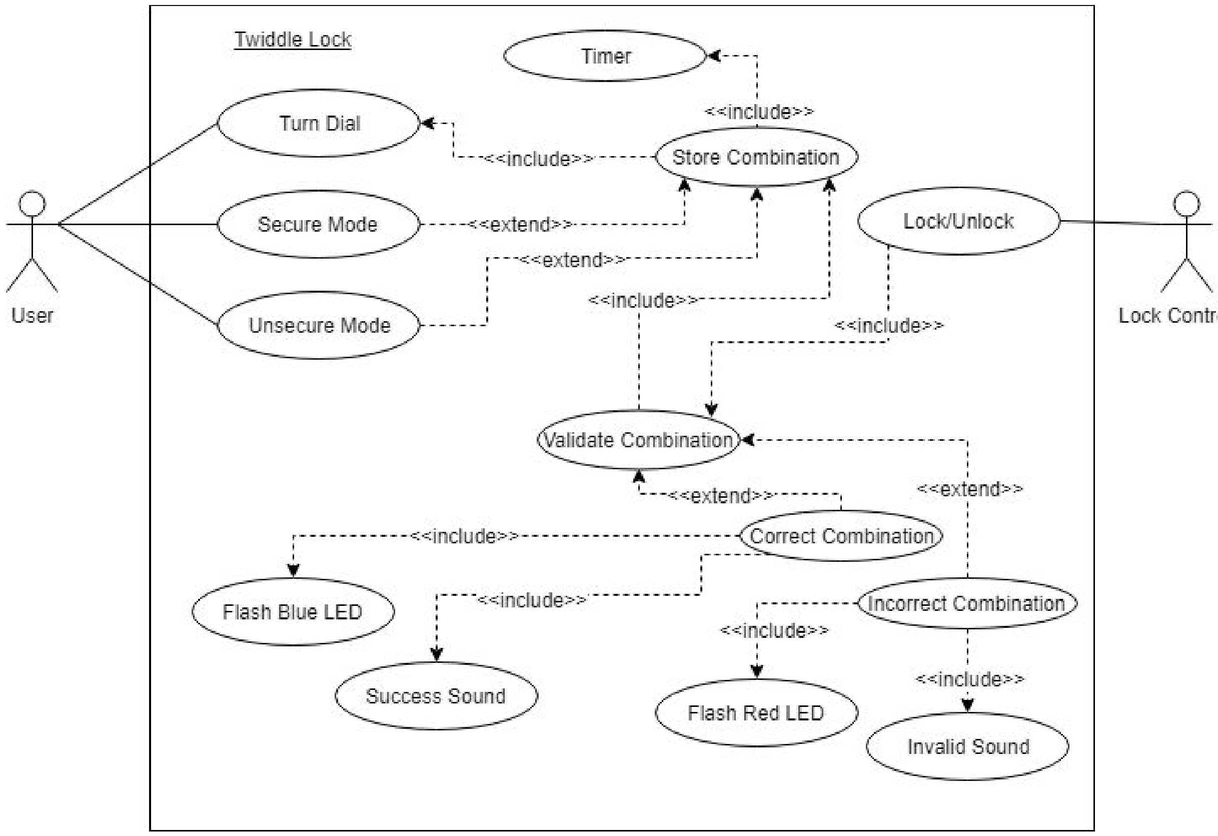


Figure 1: Implemented Use Case Model

From Figure 1, the user can choose which mode of operation they would like to use and a counter begins, if there is no input (rotation) from the dial, after a set time (5[s]), the system times out and waits for the user to press one of the mode buttons again.
Note: the user can have a live update of the combination they enter and the mode of operation if the device is connected to a display.

Specification and Design:

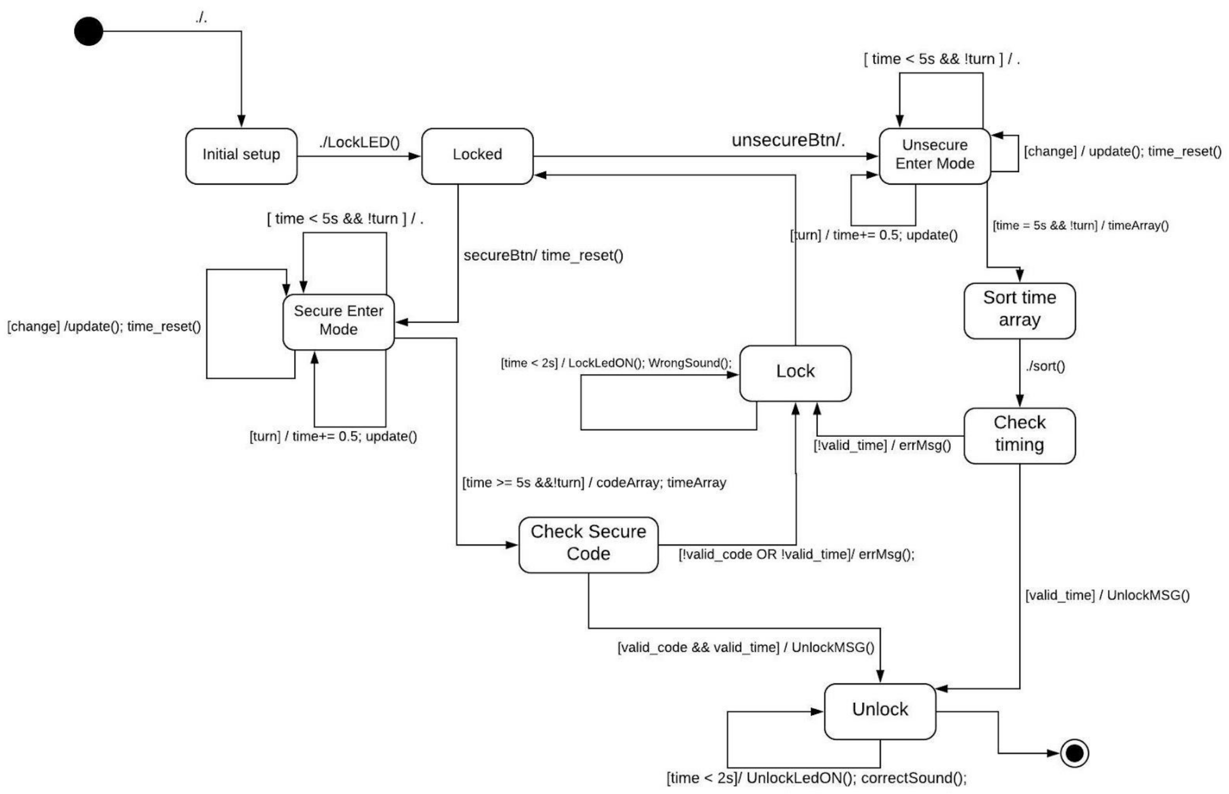

./.
Initial setup
./LockLED()
Locked
unsecureBtn/.
Unsecure Enter Mode
[time < 5s && !turn] / .
[change] / update(); time_reset()
[turn] / time+= 0.5; update()
[time = 5s && !turn] / timeArray()
Sort time array
./sort()
Check timing
[!valid_time] / errMsg()
[valid_time] / UnlockMSG()
secureBtn/ time_reset()
[time < 5s && !turn] / .
Secure Enter Mode
[change] /update(); time_reset()
[turn] / time+= 0.5; update()
[time >= 5s &&!turn] / codeArray; timeArray
Lock
[time < 2s] / LockLedON(); WrongSound();
Check Secure Code
[!valid_code OR !valid_time]/ errMsg();
[valid_code && valid_time] / UnlockMSG()
Unlock
[time < 2s]/ UnlockLedON(); correctSound();

Figure 2: UML Twiddle Lock State Chart

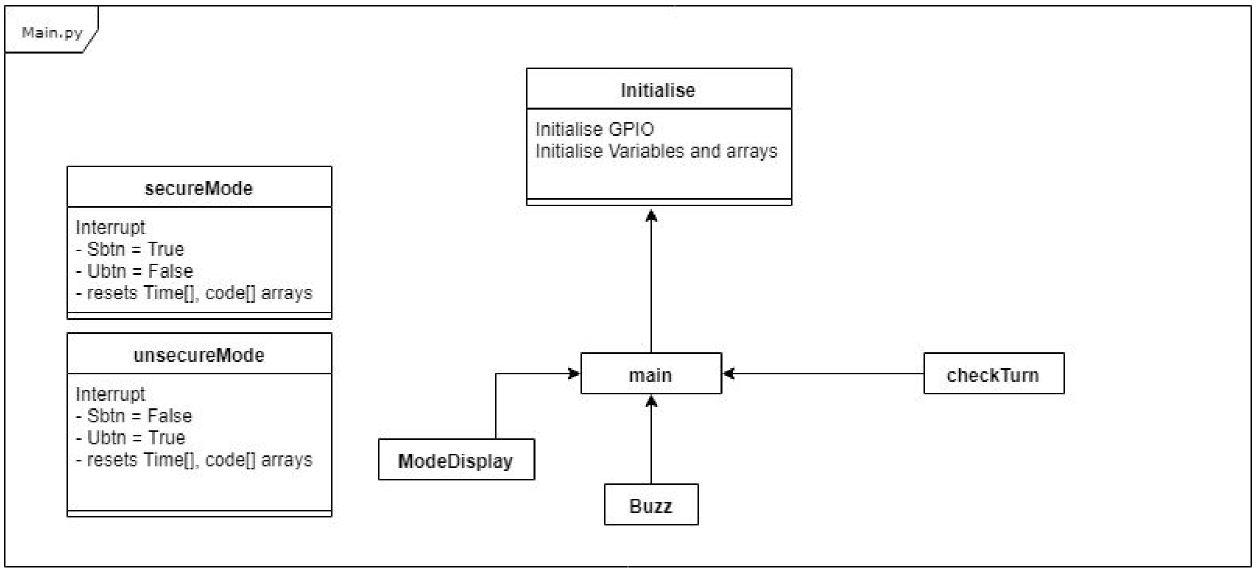


Figure 3: UML Twiddle Lock Function Diagram

From the state chart in Figure 2, it can be seen how the program is expected to behave and Figure 3, it can be seen how the functions within the Main.py file work together to form the lock controller of the system as seen in Figure 1.

Implementation:

The testing and development of the Twiddle Lock device will be done on a breadboard as seen in Figure 4 below.

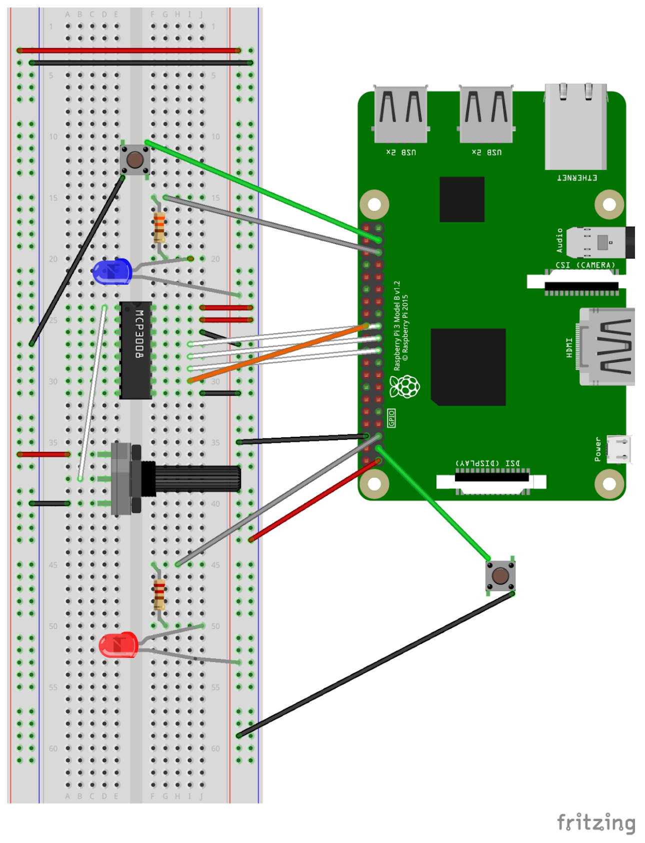
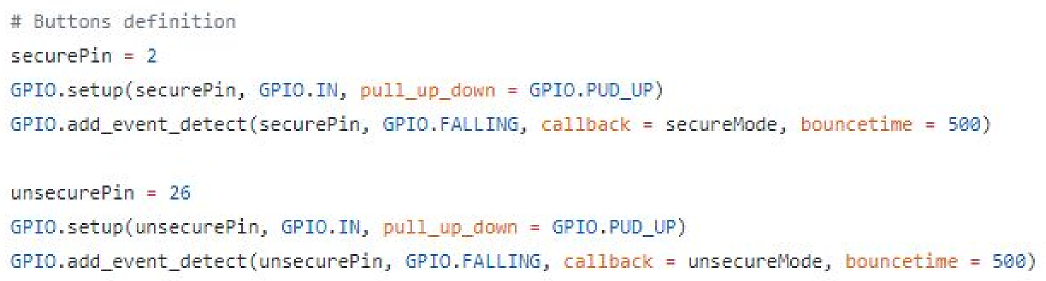


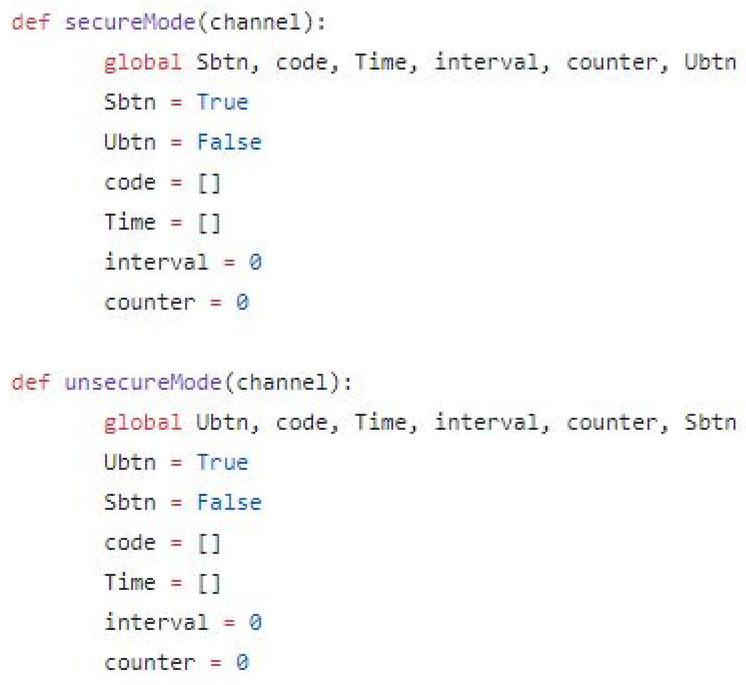
Figure 4: Hardware implementation of Twiddle Lock combination lock

In this section, important code snippets from the Main.py file (in appendix) will highlighted and explained further.

From Figure 3 above, it can be seen how the functions implemented into the system interact with each other.
Note that the functions: secureMode and unsecureMode are called after a mode selection button is pressed and are initialised as interrupts, this is seen in Figure 5 below.



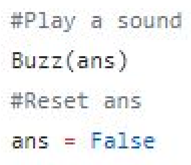
```
# Buttons definition
securePin = 2
GPIO.setup(securePin, GPIO.IN, pull_up_down = GPIO.PUD_UP)
GPIO.add_event_detect(securePin, GPIO.FALLING, callback = secureMode, bouncetime = 500)

unsecurePin = 26
GPIO.setup(unsecurePin, GPIO.IN, pull_up_down = GPIO.PUD_UP)
GPIO.add_event_detect(unsecurePin, GPIO.FALLING, callback = unsecureMode, bouncetime = 500)
```



```
def secureMode(channel):
        global Sbtn, code, Time, interval, counter, Ubtn
        Sbtn = True
        Ubtn = False
        code = []
        Time = []
        interval = 0
        counter = 0

def unsecureMode(channel):
        global Ubtn, code, Time, interval, counter, Sbtn
        Ubtn = True
        Sbtn = False
        code = []
        Time = []
        interval = 0
        counter = 0
```

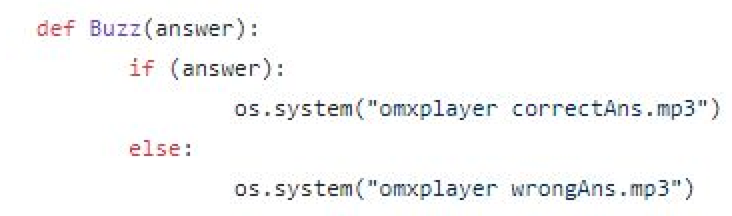
Figure 5: setup of the button interrupts (Python)

Note from Figure 5, that in every instance that a ‘mode of operation’ button is pressed that the boolean value of the buttons is set accordingly and that the input arrays (Time[] and code[]) from the dial are re-initialised (emptied). Also, the timing is reset (counter and interval). The objective with each button press is that the user is given a new and honest attempt at entering a combination into the device using the dial.



```
#Play a sound
Buzz(ans)
#Reset ans
ans = False
```

6a

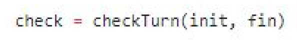


```
def Buzz(answer):
        if (answer):
                os.system("omxplayer correctAns.mp3")
        else:
                os.system("omxplayer wrongAns.mp3")
```

6b

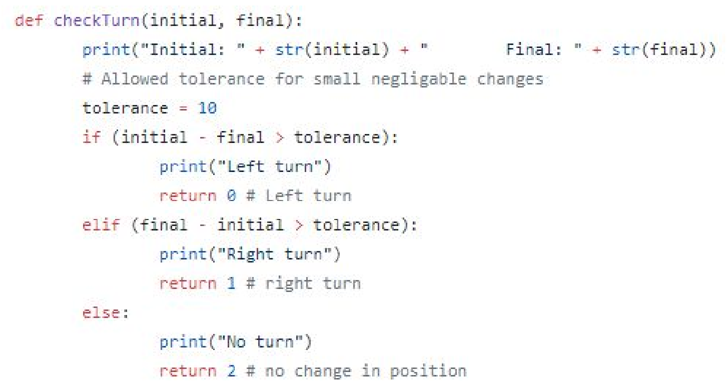
Figure 6: Implementation of sound feedback after combination is entered

In Figure 6a, after the counter has timed out, which occurs when there is no turn of the dial for a set time 5[s], the Buzz function is called with a boolean parameter called 'ans' Depending on the the value of 'ans', will result is a respective sound been played through a speaker connected to the AUX port of the Raspberry PI as seen in Figure 6b.
Note: the audio files need to be in the same directory as Main.py.



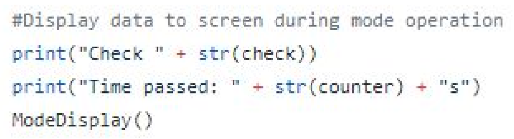
```
        check = checkTurn(init, fin)
```

7a



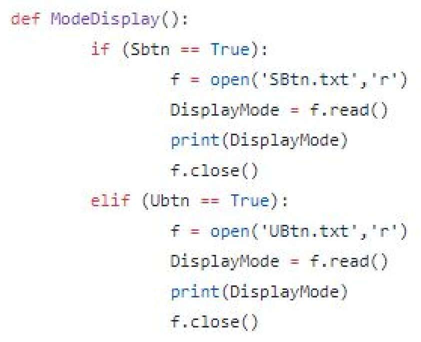
```
def checkTurn(initial, final):
        print("Initial: " + str(initial) + "        Final: " + str(final))
        # Allowed tolerance for small negligable changes
        tolerance = 10
        if (initial - final > tolerance):
                print("Left turn")
                return 0 # Left turn
        elif (final - initial > tolerance):
                print("Right turn")
                return 1 # right turn
        else:
                print("No turn")
                return 2 # no change in position
```

7b



```
#Display data to screen during mode operation
print("Check " + str(check))
print("Time passed: " + str(counter) + "s")
ModeDisplay()
```

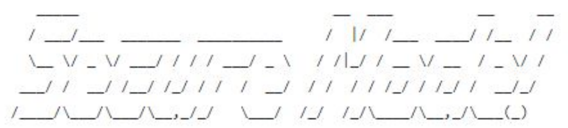
7c



```
def ModeDisplay():
        if (Sbtn == True):
                f = open('SBtn.txt','r')
                DisplayMode = f.read()
                print(DisplayMode)
                f.close()
        elif (Ubtn == True):
                f = open('UBtn.txt','r')
                DisplayMode = f.read()
                print(DisplayMode)
                f.close()
```

7d

7e



7f

Figure 7: Implementation of the data that is displayed once on of the mode buttons have been pressed (data is displayed to a screen).

Figure 7a and 7c identifies what data is displayed to the screen once a mode (secure/ unsecure) button has been pressed, Figure 7b indicates how the user will receive feedback with respect to the direction of the turn (left/ right or no turn).

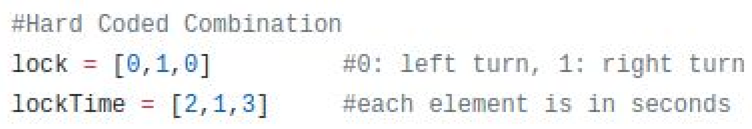
The first print statement in Figure 7c will indicate if the turn of the dial is used by the program correctly as this is the value that will be appended into the the code[] array; as such the value of variable 'check' can be 0,1 or 2 which corresponds to left, right, or no turn respectively.

The second print statement in Figure 7c indicates the time that has elapsed during the current turn of the dial. Once the turn of the dial has changed in direction (value of 'check') the timer is reset. This timer value is appended into the Time[] array.

In Figure 7c, a function ModeDisplay() is called and in Figure 6d, ModeDisplay() is defined, this function requires two .txt files that will display the mode of operation after the respective button is pressed as seen in figures 7e and 7f.

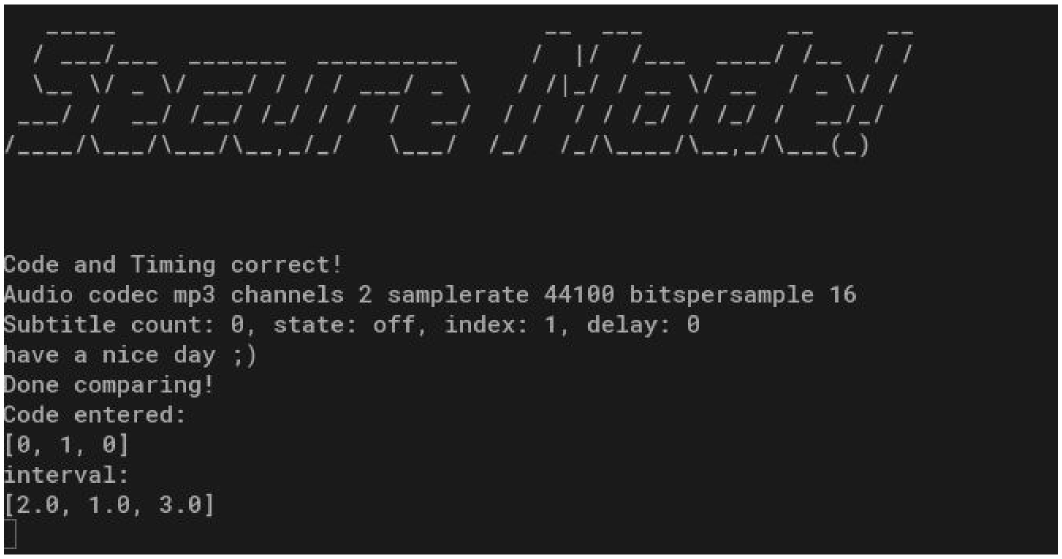
Note that the implementation of figure 7 does not affect the function of the device from as seen in Figure 1, however this does make the testing and demo of the device more convenient.

Validation and Performance:



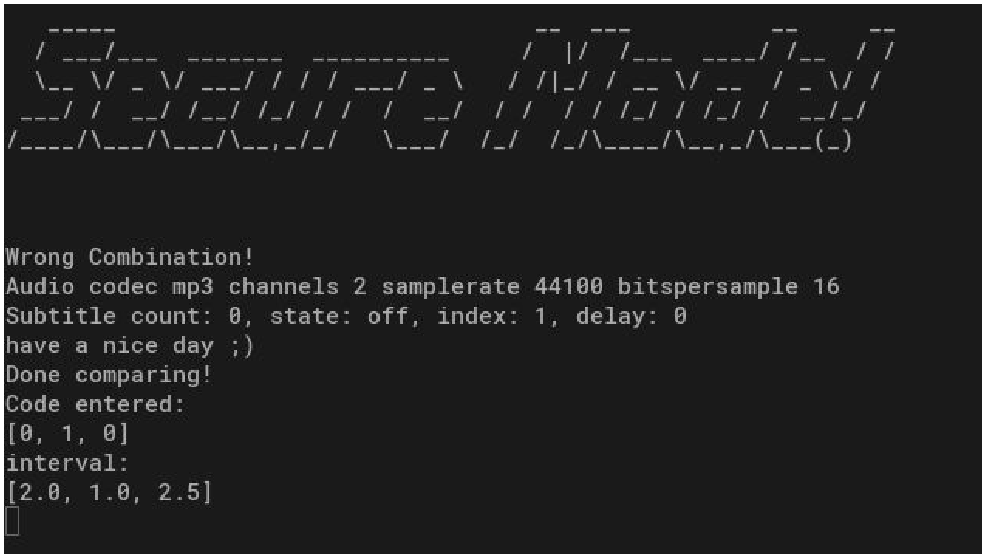
```
#Hard Coded Combination
lock = [0,1,0]          #0: left turn, 1: right turn
lockTime = [2,1,3]      #each element is in seconds
```

Figure 8: The hard coded values required to open the lock



```
   _____                             __  ___          __     __
  / ___/___  _______  __________    /  |/  /___  ____/ /__  / /
  \__ \/ _ \/ ___/ / / / ___/ _ \  / /|_/ / __ \/ __  / _ \/ /
 ___/ /  __/ /__/ /_/ / /  /  __/ / /  / / /_/ / /_/ /  __/_/
/____/\___/\___/\__,_/_/   \___/ /_/  /_/\____/\__,_/\___(_)



Code and Timing correct!
Audio codec mp3 channels 2 samplerate 44100 bitspersample 16
Subtitle count: 0, state: off, index: 1, delay: 0
have a nice day ;)
Done comparing!
Code entered:
[0, 1, 0]
interval:
[2.0, 1.0, 3.0]
```

Figure 9: Result of entering the correct code and time interval in secure mode

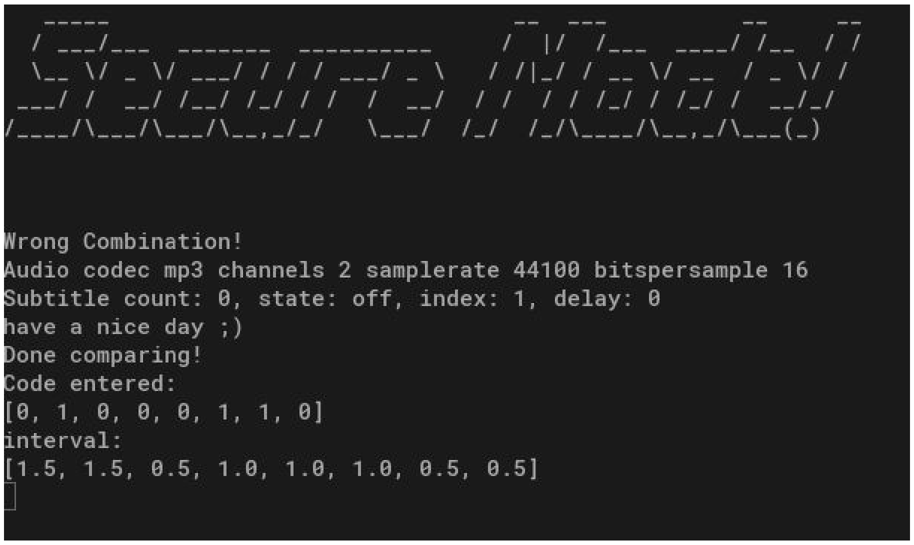


```
   _____                             __  ___          __     __
  / ___/___  _______  __________    /  |/  /___  ____/ /__  / /
  \__ \/ _ \/ ___/ / / / ___/ _ \  / /|_/ / __ \/ __  / _ \/ /
 ___/ /  __/ /__/ /_/ / /  /  __/ / /  / / /_/ / /_/ /  __/_/
/____/\___/\___/\__,_/_/   \___/ /_/  /_/\____/\__,_/\___(_)



Wrong Combination!
Audio codec mp3 channels 2 samplerate 44100 bitspersample 16
Subtitle count: 0, state: off, index: 1, delay: 0
have a nice day ;)
Done comparing!
Code entered:
[0, 1, 0]
interval:
[2.0, 1.0, 2.5]
```

Figure 10: Result of entering the correct code but the wrong time interval in secure mode

```
   _____                             __  ___          __     __
  / ___/___  _______  __________    /  |/  /___  ____/ /__  / /
  \__ \/ _ \/ ___/ / / / ___/ _ \  / /|_/ / __ \/ __  / _ \/ /
 ___/ /  __/ /__/ /_/ / /  /  __/ / /  / / /_/ / /_/ /  __/_/
/____/\___/\___/\__,_/_/   \___/ /_/  /_/\____/\__,_/\___(_)
```

Figure 11: Result of entering the correct time interval but the wrong code in secure mode



```
  _____                              __  ___          __     __
 / ___/___  _______  __________     /  |/  /___  ____/ /__  / /
 \__ \/ _ \/ ___/ / / / ___/ _ \   / /|_/ / __ \/ __  / _ \/ /
___/ /  __/ /__/ /_/ / /  /  __/  / /  / / /_/ / /_/ /  __/_/
/____/\___/\___/\__,_/_/   \___/  /_/  /_/\____/\__,_/\___(_)



Wrong Combination!
Audio codec mp3 channels 2 samplerate 44100 bitspersample 16
Subtitle count: 0, state: off, index: 1, delay: 0
have a nice day ;)
Done comparing!
Code entered:
[0, 1, 0, 0, 0, 1, 1, 0]
interval:
[1.5, 1.5, 0.5, 1.0, 1.0, 1.0, 0.5, 0.5]
```

Figure 12: Result of entering the wrong code and the wrong time interval in secure mode

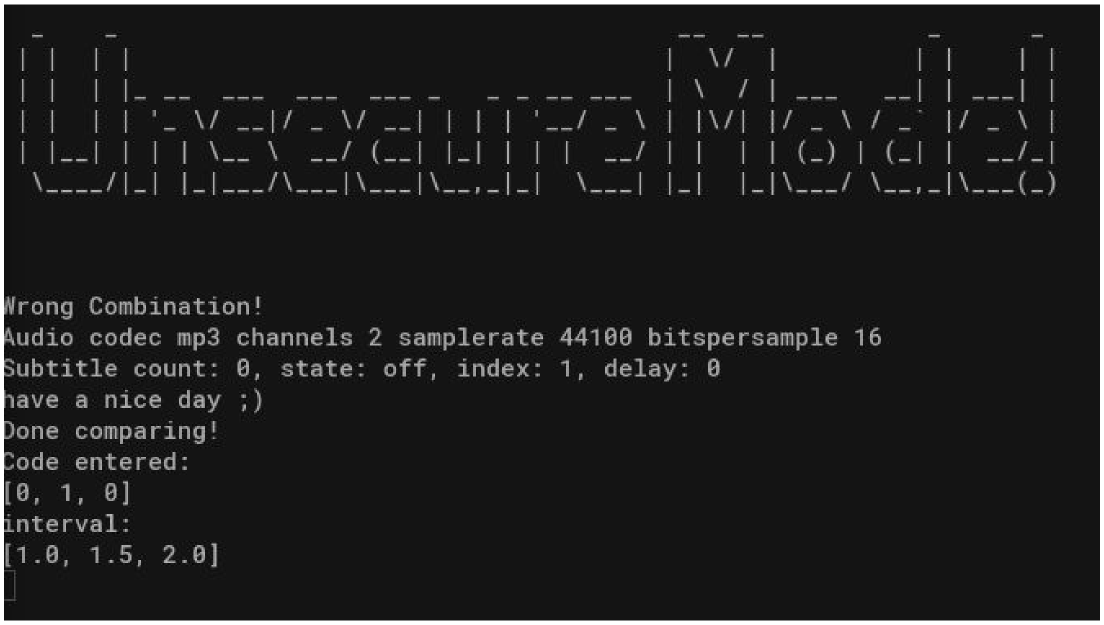
In secure mode the only way to send the unlock signal is to enter both the correct code (the correct order and number of directions) and the correct time taken for each turn. From the figures above (Figure 8 to 11) we get the outputs of the system for different input combinations which are tabulated down below in Table 1.

Input:	Resultant State:

Wrong code and wrong time intervals	Locked
Correct code but wrong time intervals	Locked
Correct time intervals but wrong code	Locked
Correct code and time intervals	Unlocked

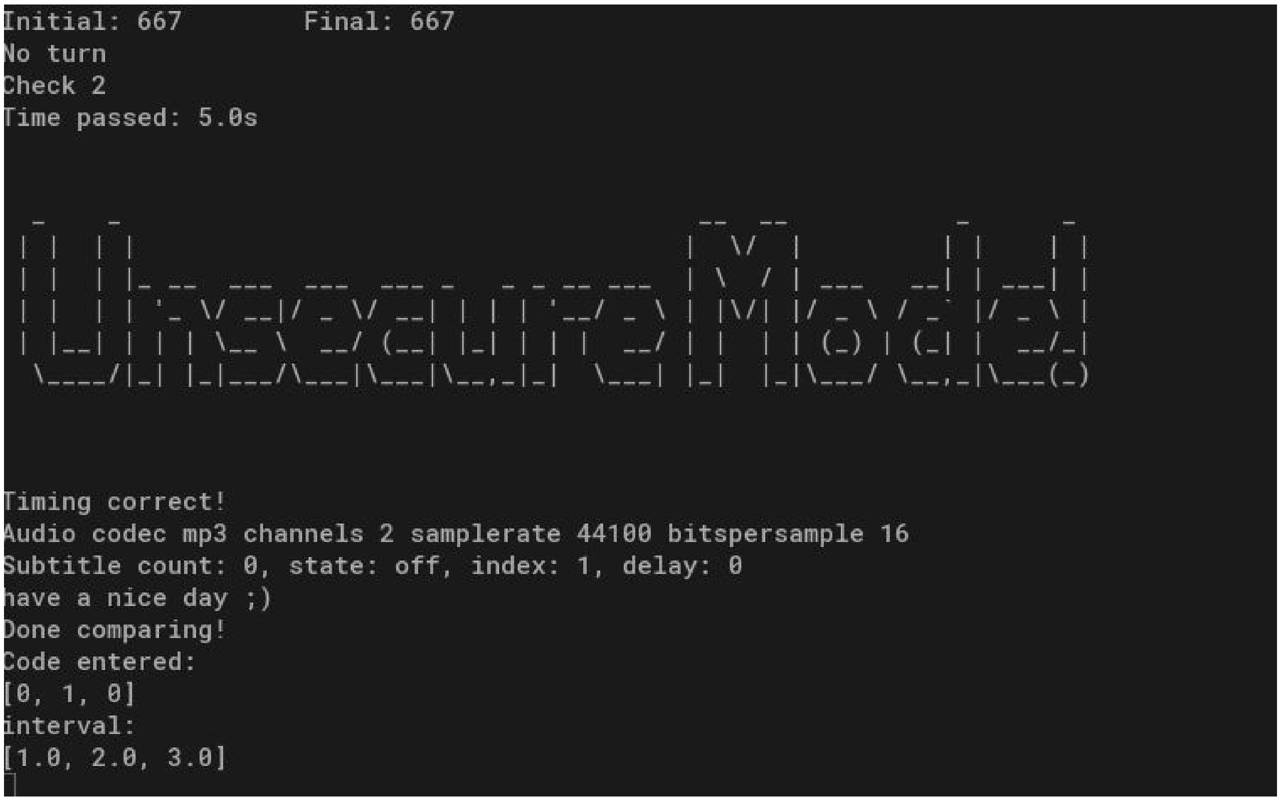
Table 1: The resultant states for different input combinations in **secure mode**

From the results shown, the only way to successfully produce the unlock signal is to get both the direction and time taken for each turn as required.



```
 _    _                                        __  __           _      _ 
| |  | |                                      |  \/  |         | |    | |
| |  | |_ __  ___  ___  ___ _   _ _ __ ___    | \  / | ___   __| | ___| |
| |  | | '_ \/ __|/ _ \/ __| | | | '__/ _ \   | |\/| |/ _ \ / _` |/ _ \ |
| |__| | | | \__ \  __/ (__| |_| | | |  __/   | |  | | (_) | (_| |  __/_|
 \____/|_| |_|___/\___|\___|\__,_|_|  \___|   |_|  |_|\___/ \__,_|\___(_)



Wrong Combination!
Audio codec mp3 channels 2 samplerate 44100 bitspersample 16
Subtitle count: 0, state: off, index: 1, delay: 0
have a nice day ;)
Done comparing!
Code entered:
[0, 1, 0]
interval:
[1.0, 1.5, 2.0]
```

Figure 13: Result of entering the wrong time interval and correct code in unsecure mode



```
Initial: 667       Final: 667
No turn
Check 2
Time passed: 5.0s


  _    _                                      __  __           _      _ 
 | |  | |                                    |  \/  |         | |    | |
 | |  | |_ __  ___  ___  ___ _   _ _ __ ___  | \  / | ___   __| | ___| |
 | |  | | '_ \/ __|/ _ \/ __| | | | '__/ _ \ | |\/| |/ _ \ / _` |/ _ \ |
 | |__| | | | \__ \  __/ (__| |_| | | |  __/ | |  | | (_) | (_| |  __/_|
  \____/|_| |_|___/\___|\___|\__,_|_|  \___| |_|  |_|\___/ \__,_|\___(_)


Timing correct!
Audio codec mp3 channels 2 samplerate 44100 bitspersample 16
Subtitle count: 0, state: off, index: 1, delay: 0
have a nice day ;)
Done comparing!
Code entered:
[0, 1, 0]
interval:
[1.0, 2.0, 3.0]
```

Figure 14: Result of entering the correct time interval (order doesn't matter) in unsecure mode

In unsecure mode the only way to send the unlock signal is to enter the correct time taken for each turn where the order for each timing doesn't matter. From the figures above (Figure 8, 13 and 14) we get the outputs of the system for different input combinations which are tabulated down below in Table 2.

Input:	Resultant State:
Wrong code and wrong time intervals	Locked
Correct code but wrong time intervals	Locked
Correct time intervals but wrong code	Unlocked
Correct code and time intervals	Unlocked

Table 2: The resultant states for different input combinations in **unsecure mode**

From the results shown, the lock signal can be produced if the time taken for each turn is correct and the order of the inputs does not matter, as a result the unlock signal is independent from the direction of turns entered by the user.

Performance:

The timing of the the rotation of the dial seems suitably accurate as well as consistent and therefore usable as the device uses the time library in python (t.sleep(0.5)). It is important to note that readings are taken at 500[ms] intervals, which allows enough time for the user to input a combination without having to accommodate for unnecessary tolerances while turning the dial.

Conclusion:

Ultimately the device can be considered successful in its operation, as the Twiddle Lock operates as per most of the users requirements, where the product deviated from the users requirements, valid arguments or reasons have been given.

From the testing done in the Validation and Performance section above it is clear that secure mode is safer than in unsecure mode, as the sequence and duration of the input from the dial has to match given values in secure mode, while any sequence of durations from the dial that matches a given duration array constitutes as a complete and successful combination in unsecure mode.
The difference in time taken to complete both modes of operations is negligible.

However, there is room for improvement, firstly the device can be made more secure by encrypting the required combination instead of hardcoding these values in the form of arrays into the Main.py file. Also, allowing the user to bypass the secure mode by selecting the unsecure button would not be suitable in the real world.
By implementing a GUI for the user, the user could interact with the device in a more user friendly manner.
Lastly, the time taken between the last turn of the dial and until the user can re-enter another combination could be shortened; as this could be considered safety feature, we decided against implementing this change.

Therefore the ‘Twiddle lock’ can be implemented into a useful real world product especially if the security flaws mentioned above are fixed.

References:

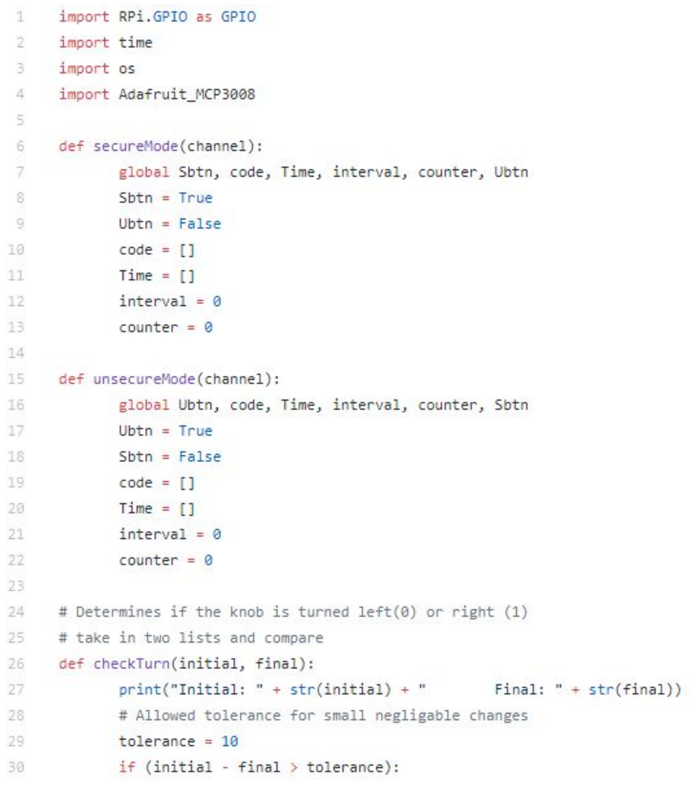
Ibanez, R. and Pervin, W. (2017). *RASPBERRY PI ASSEMBLER*. 1st ed. Mountain View: McGraw-Hill Custom Publishing, pp.72-75.

The Github Repository of this project:

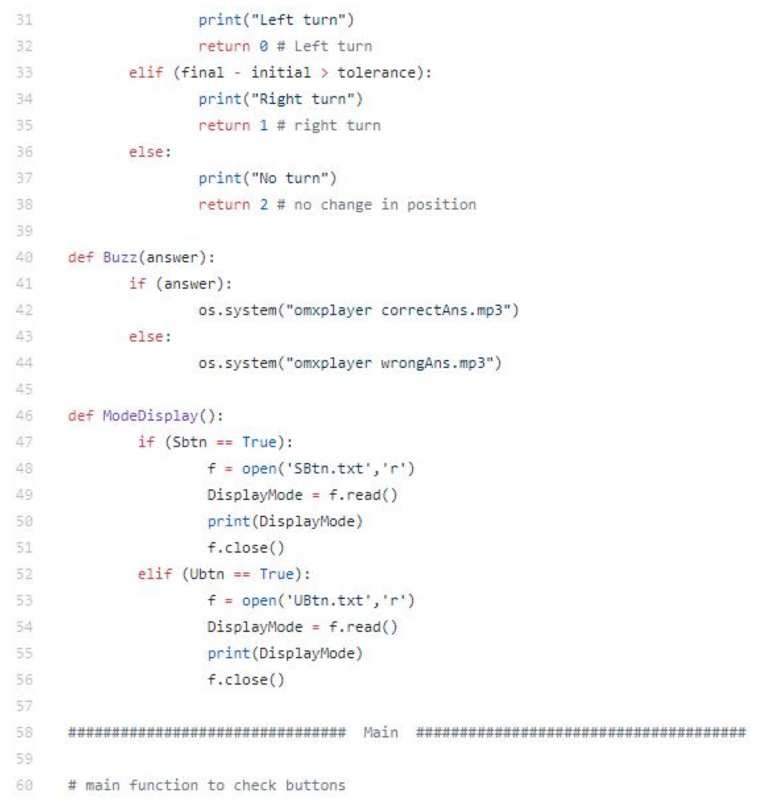
Kukreja, S. and Olivier, J. (2018). *itssnehin/TwiddleLockRPi3*. [online] GitHub. Available at: https://github.com/itssnehin/TwiddleLockRPi3/tree/master [Accessed 23 Oct. 2018].

Appendix:

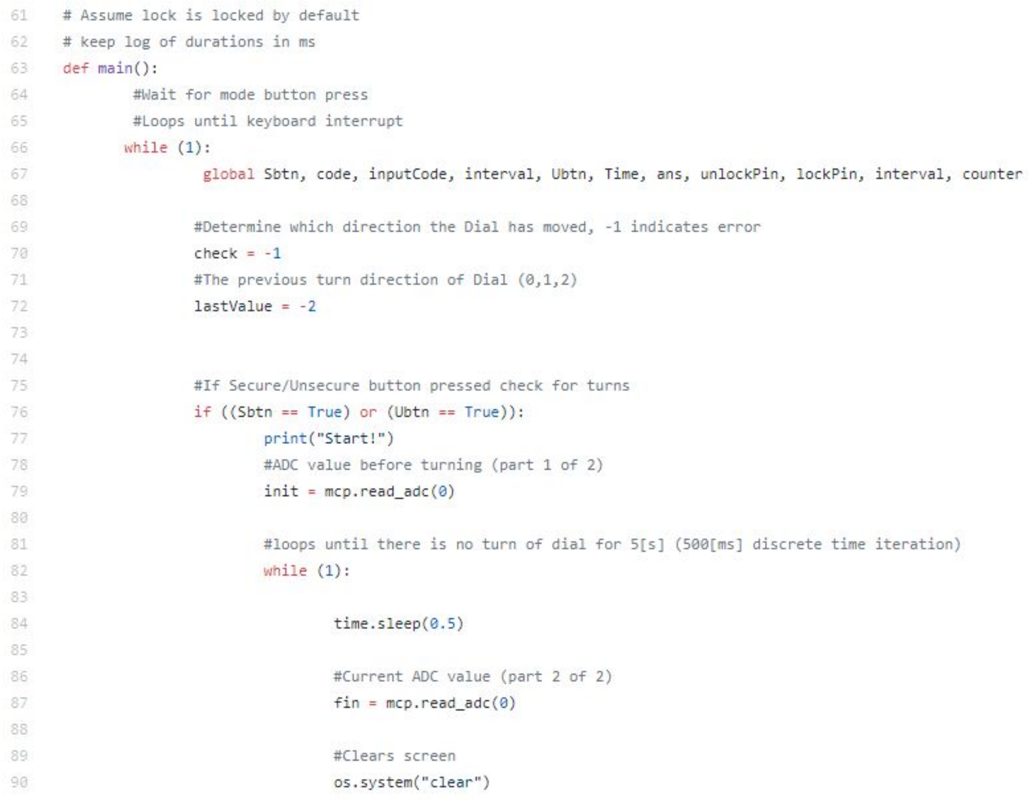
Main.py



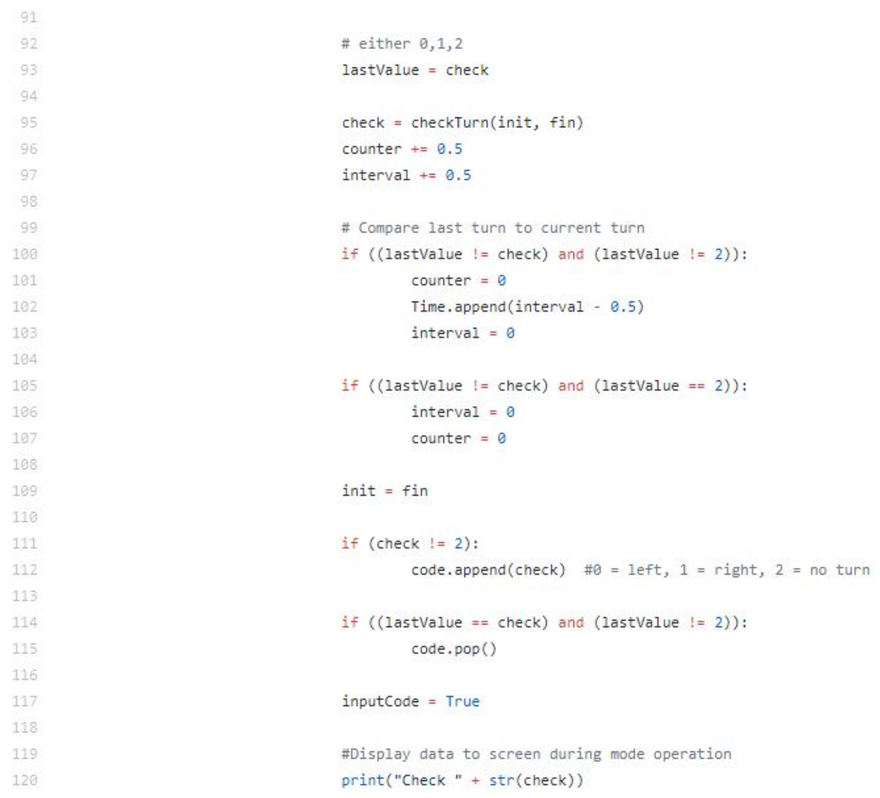
```python
import RPi.GPIO as GPIO
import time
import os
import Adafruit_MCP3008

def secureMode(channel):
	global Sbtn, code, Time, interval, counter, Ubtn
	Sbtn = True
	Ubtn = False
	code = []
	Time = []
	interval = 0
	counter = 0

def unsecureMode(channel):
	global Ubtn, code, Time, interval, counter, Sbtn
	Ubtn = True
	Sbtn = False
	code = []
	Time = []
	interval = 0
	counter = 0

# Determines if the knob is turned left(0) or right (1)
# take in two lists and compare
def checkTurn(initial, final):
	print("Initial: " + str(initial) + "        Final: " + str(final))
	# Allowed tolerance for small negligable changes
	tolerance = 10
	if (initial - final > tolerance):
```



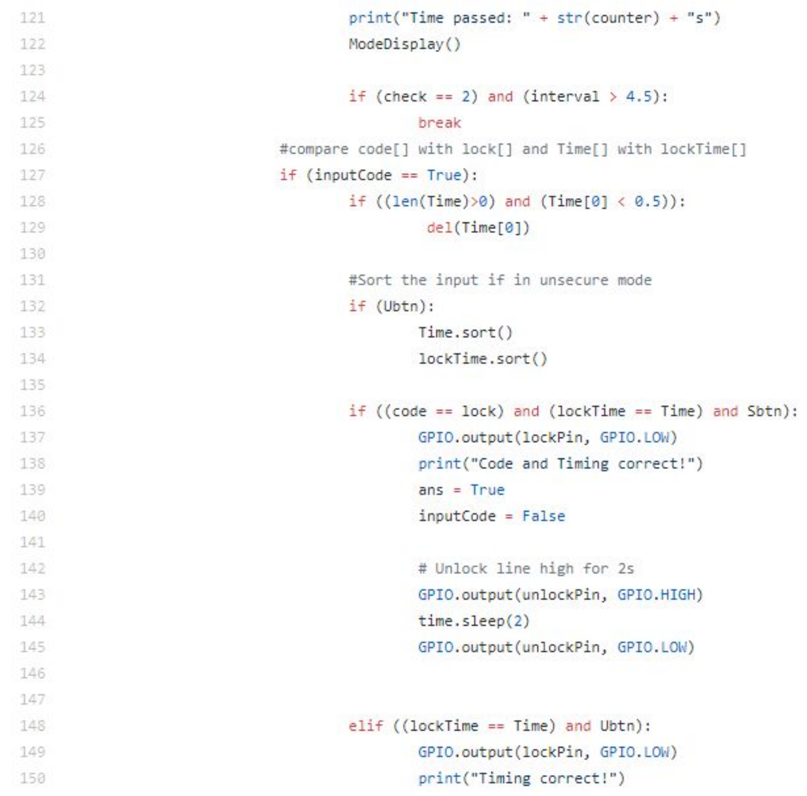
```
                print("Left turn")
                return 0 # Left turn
        elif (final - initial > tolerance):
                print("Right turn")
                return 1 # right turn
        else:
                print("No turn")
                return 2 # no change in position

def Buzz(answer):
        if (answer):
                os.system("omxplayer correctAns.mp3")
        else:
                os.system("omxplayer wrongAns.mp3")

def ModeDisplay():
         if (Sbtn == True):
                 f = open('SBtn.txt','r')
                 DisplayMode = f.read()
                 print(DisplayMode)
                 f.close()
         elif (Ubtn == True):
                 f = open('UBtn.txt','r')
                 DisplayMode = f.read()
                 print(DisplayMode)
                 f.close()

################################  Main  ######################################

# main function to check buttons
```



```
# Assume lock is locked by default
# keep log of durations in ms
def main():
        #Wait for mode button press
        #Loops until keyboard interrupt
        while (1):
                 global Sbtn, code, inputCode, interval, Ubtn, Time, ans, unlockPin, lockPin, interval, counter

                #Determine which direction the Dial has moved, -1 indicates error
                check = -1
                #The previous turn direction of Dial (0,1,2)
                lastValue = -2


                #If Secure/Unsecure button pressed check for turns
                if ((Sbtn == True) or (Ubtn == True)):
                        print("Start!")
                        #ADC value before turning (part 1 of 2)
                        init = mcp.read_adc(0)

                        #loops until there is no turn of dial for 5[s] (500[ms] discrete time iteration)
                        while (1):

                                time.sleep(0.5)

                                #Current ADC value (part 2 of 2)
                                fin = mcp.read_adc(0)

                                #Clears screen
                                os.system("clear")
```

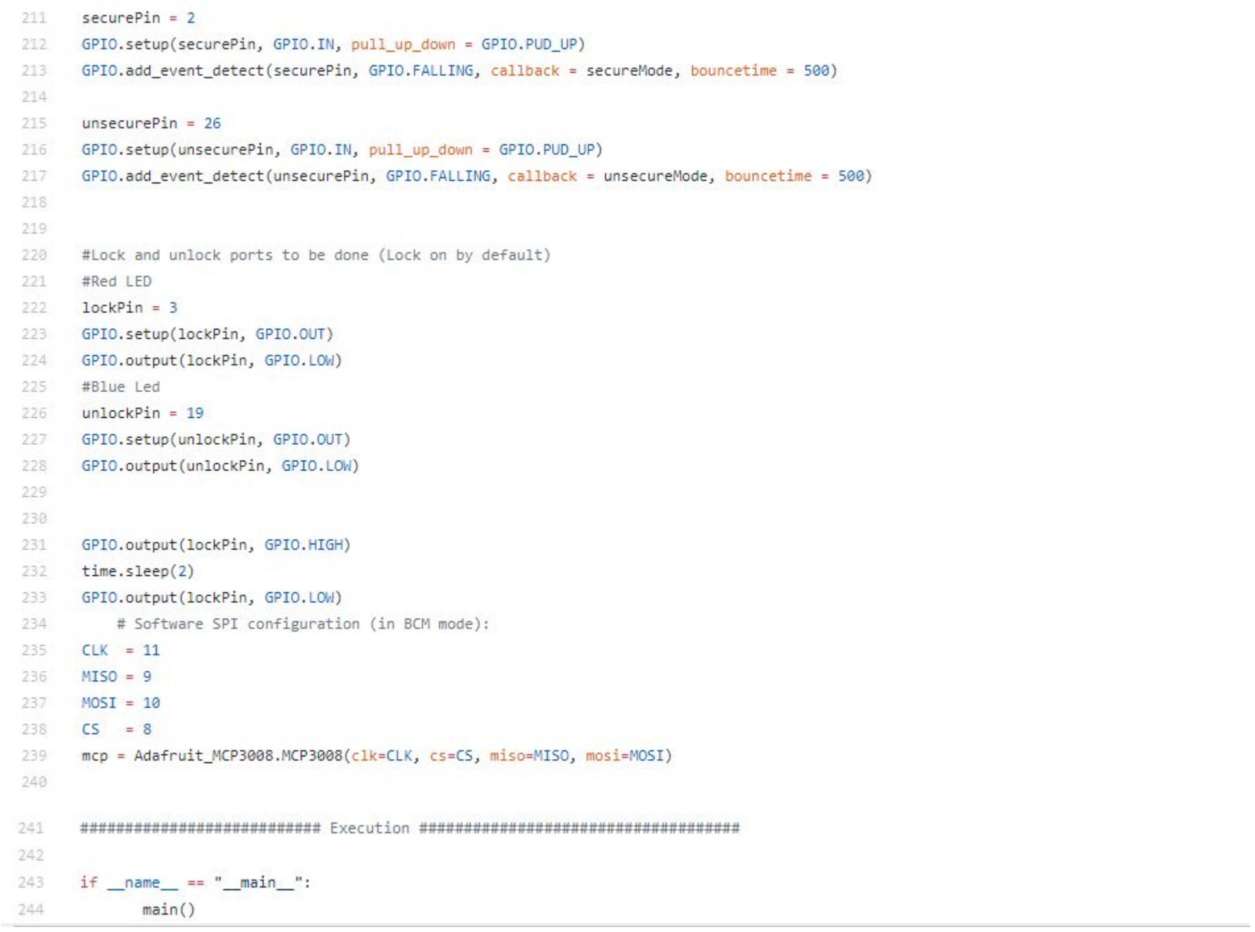


```python

            # either 0,1,2
            lastValue = check

            check = checkTurn(init, fin)
            counter += 0.5
            interval += 0.5

            # Compare last turn to current turn
            if ((lastValue != check) and (lastValue != 2)):
                counter = 0
                Time.append(interval - 0.5)
                interval = 0

            if ((lastValue != check) and (lastValue == 2)):
                interval = 0
                counter = 0

            init = fin

            if (check != 2):
                code.append(check)  #0 = left, 1 = right, 2 = no turn

            if ((lastValue == check) and (lastValue != 2)):
                code.pop()

            inputCode = True

            #Display data to screen during mode operation
            print("Check " + str(check))
```



```python
            print("Time passed: " + str(counter) + "s")
            ModeDisplay()

            if (check == 2) and (interval > 4.5):
                break
        #compare code[] with lock[] and Time[] with lockTime[]
        if (inputCode == True):
            if ((len(Time)>0) and (Time[0] < 0.5)):
                 del(Time[0])

            #Sort the input if in unsecure mode
            if (Ubtn):
                Time.sort()
                lockTime.sort()

            if ((code == lock) and (lockTime == Time) and Sbtn):
                GPIO.output(lockPin, GPIO.LOW)
                print("Code and Timing correct!")
                ans = True
                inputCode = False

                # Unlock line high for 2s
                GPIO.output(unlockPin, GPIO.HIGH)
                time.sleep(2)
                GPIO.output(unlockPin, GPIO.LOW)


            elif ((lockTime == Time) and Ubtn):
                GPIO.output(lockPin, GPIO.LOW)
                print("Timing correct!")
```

```
                    ans = True
                    inputCode = False

                    GPIO.output(unlockPin, GPIO.HIGH)
                    time.sleep(2)
                    GPIO.output(unlockPin, GPIO.LOW)

                else:
                    ans = False
                    print("Wrong Combination!")

                    GPIO.output(lockPin, GPIO.HIGH)
                    time.sleep(2)
                    GPIO.output(lockPin, GPIO.LOW)

                #Play a sound
                Buzz(ans)
                #Reset ans
                ans = False



            Sbtn = False
            Ubtn = False
            print("Done comparing!")
            print("Code entered: ")
            print(code)
            print("interval: ")
            print(Time)
            time.sleep(1.5)


########################## Initial Setup #################################

#Global init

#Dynamic Arrays of the combination inserted by the user
Time = []
code = []

#Hard Coded Combination
lock = [0,1,0,1]       #0: left turn, 1: right turn
lockTime = [2,1,3,1]   #each element is in seconds

#Secure button
Sbtn = False
#Unsecure button
Ubtn = False

#Elapsed time
counter = 0
interval = 0

ans = False
inputCode  =  False

#Raspberry setup
GPIO.setmode(GPIO.BCM)

# Initialise buttons
```



```
securePin = 2
GPIO.setup(securePin, GPIO.IN, pull_up_down = GPIO.PUD_UP)
GPIO.add_event_detect(securePin, GPIO.FALLING, callback = secureMode, bouncetime = 500)

unsecurePin = 26
GPIO.setup(unsecurePin, GPIO.IN, pull_up_down = GPIO.PUD_UP)
GPIO.add_event_detect(unsecurePin, GPIO.FALLING, callback = unsecureMode, bouncetime = 500)


#Lock and unlock ports to be done (Lock on by default)
#Red LED
lockPin = 3
GPIO.setup(lockPin, GPIO.OUT)
GPIO.output(lockPin, GPIO.LOW)
#Blue Led
unlockPin = 19
GPIO.setup(unlockPin, GPIO.OUT)
GPIO.output(unlockPin, GPIO.LOW)


GPIO.output(lockPin, GPIO.HIGH)
time.sleep(2)
GPIO.output(lockPin, GPIO.LOW)
    # Software SPI configuration (in BCM mode):
CLK  = 11
MISO = 9
MOSI = 10
CS   = 8
mcp = Adafruit_MCP3008.MCP3008(clk=CLK, cs=CS, miso=MISO, mosi=MOSI)

########################### Execution ####################################

if __name__ == "__main__":
       main()
```