# EEE3096S: Practical 6/Mini-Project
# Twiddle-Lock System

23 October 2018

By:
Aadam Osman (OSMAAD003)
Kishen Patel (PTLKIS002)

**Name #1: Aadam Osman**

**Student Number #1: OSMAAD003**

**Name #2: Kishen Patel**

**Student Number #2: PTLKIS002**

**Course: EEE3096S**

**Declaration**

I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.

I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.

This report is my own work.

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.

**Signature** [illegible]　　　　**Signature** [illegible]

**Date** 23/10/2018　　　　**Date** 23/10/2018

## Table of Contents

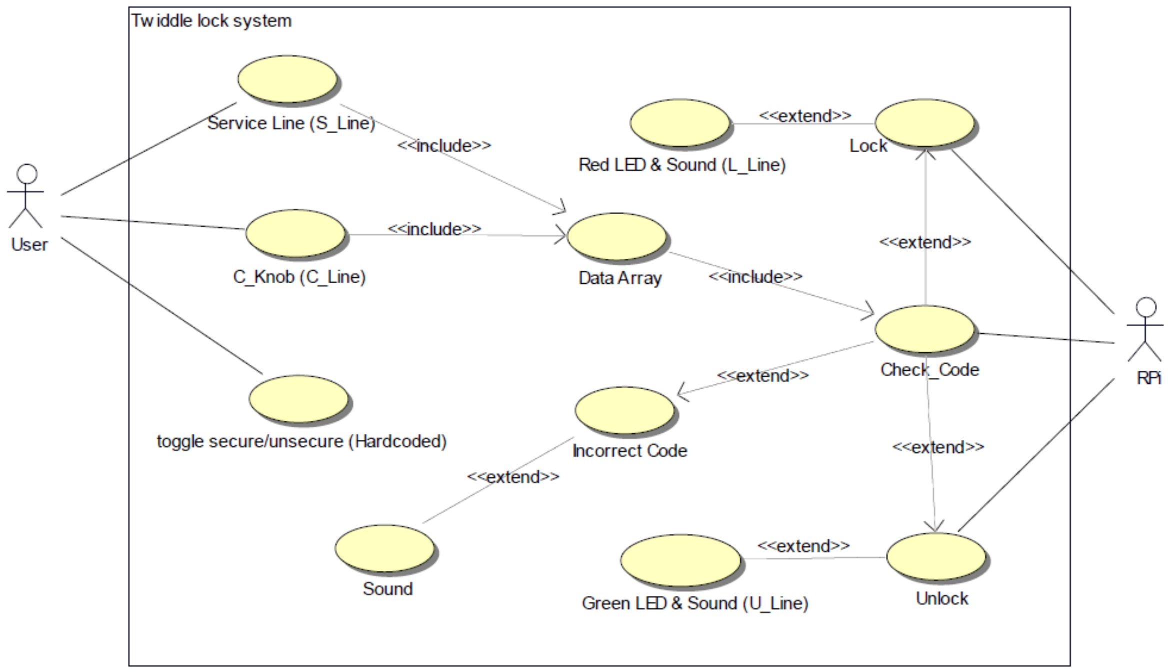
## Introduction

For the given assignment, the team was tasked in designing, building and programming (in python) a simulated twiddle lock (which is simply a Cyber-physical system) with the following function description. Given a hardcoded combination (set by the design team) the user must dial the combination (using a given rotary potentiometer) correctly within a specified time per combination key. Should the combination be correct an audio indicating a successful code entry will be initiated and the system will unlock, or lock itself depending on its previous state; and the U-line/L-line (described in the brief) will go high for 2 seconds. However, should the user input the incorrect combination, or does not meet the timing requirements the system will initiate an audio that indicates an unsuccessful code entry. It must be noted that for simulation purposes that U-line and L-line were simulated with the use of LEDs and it must be noted that on initializing the program, the system goes into a Lock state. The above stated scenario description is for one of two modes in which the system takes on, which are secure and unsecure mode. In secure mode, a correct combination entry depends on both the time it takes to turn the dial to each individual code combination and the overall code combination; whereas in unsecure mode the correct combination only depends on the time it takes to turn the dial to any state.

The sequence of design (to which will be outlined in this report) is firstly the analysis of the user requirements (of which the User Case diagram will be formulated). Given the User Case diagram the State Chart (simply describing the system operation) will be formulated along with a class diagram for the protentional python program layout. After this, discussion of how the previous design procedures/structures were used in writing the code for the given embedded system will be provided, finally concluding with validating the coded system. It must be noted in addition to designing, building and programming the above system, the task of coding a simple selection sort in ARM assembly was also briefed and the code and a small summary description can be found in Appendix B.

## Requirements

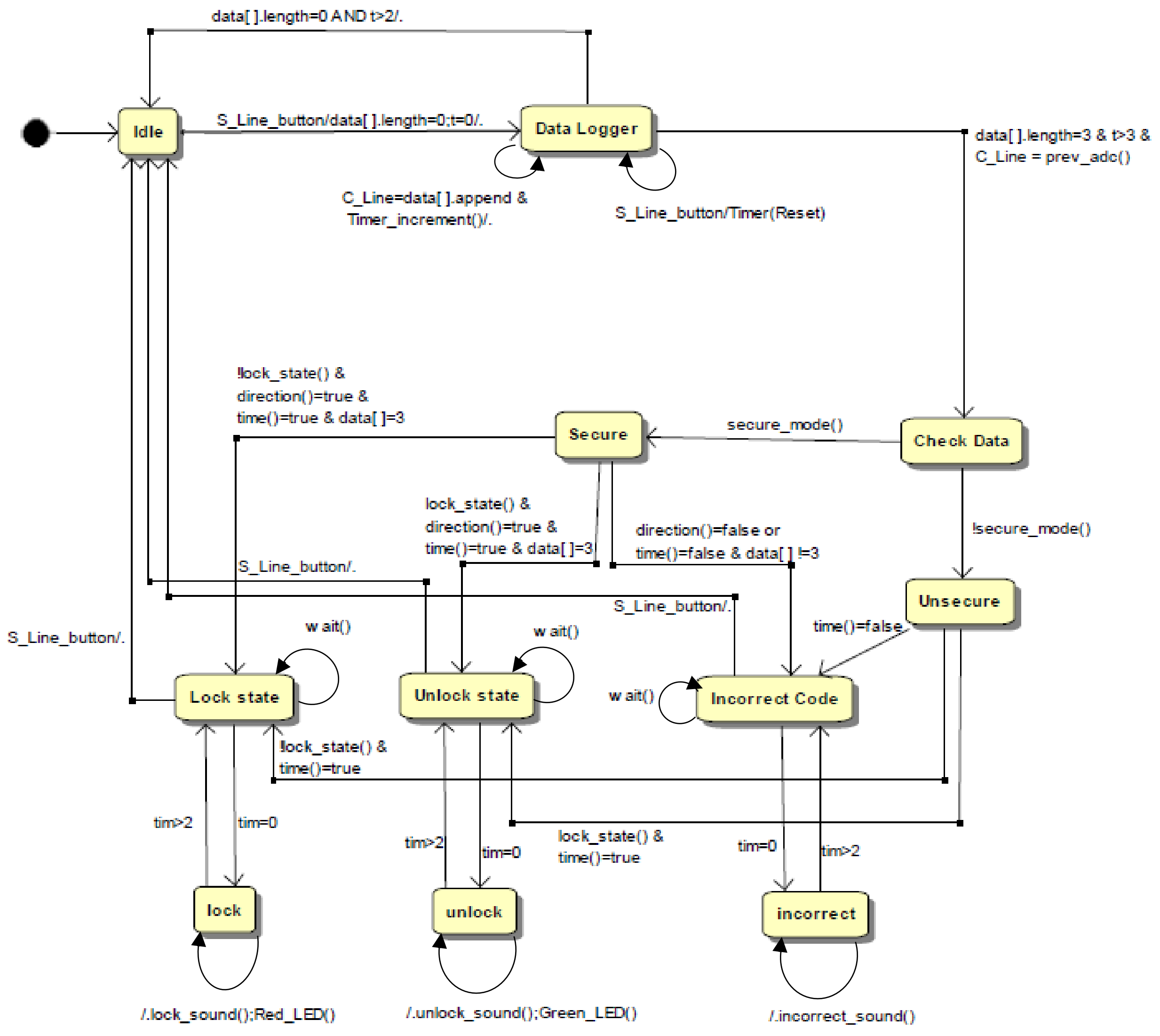
The refined UML Use Case diagram is provided below:



The basic system requirements consists of the user indicating when the system should start recording the 'twiddle knob' data which is then compared with the required code to either lock or unlock the system. The lock sequence plays a specific sound and indicates a red light whereas the unlocking sequence plays a different sound and indicates a green light. The basic requirements remain the same as the original project description. The following additions were made which includes the Data Array, where the knob position and time are recorded in arrays, and Check Code, which evaluates the data in the Data Array and decides whether to lock, unlock or keep the system in the same state.

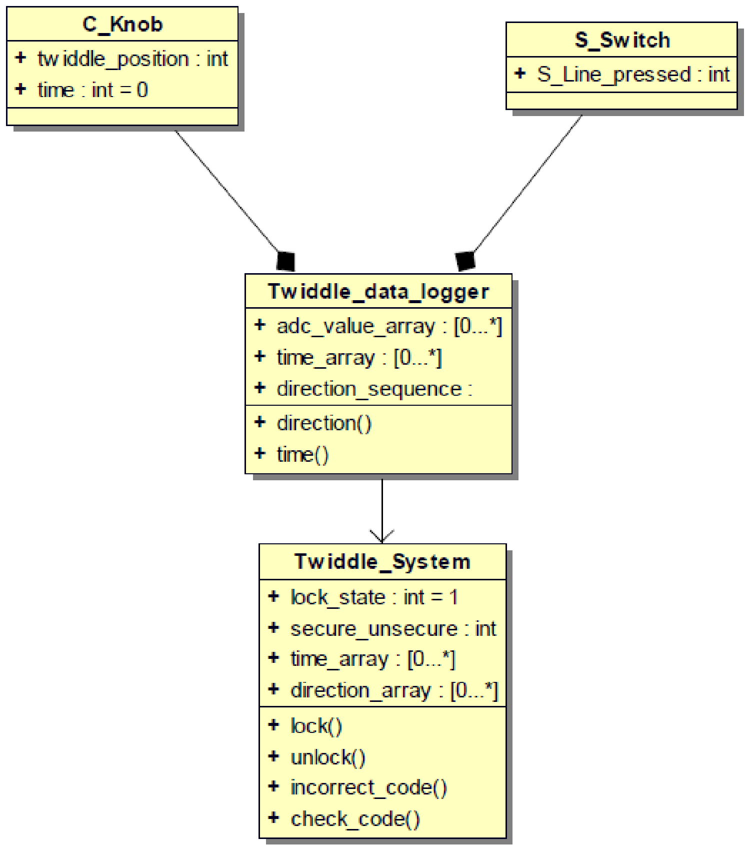
## Specifications and Design

The following diagram represents a User State chart for the implemented system.



data[ ].length=0 AND t>2/.
Idle
S_Line_button/data[ ].length=0;t=0/.
Data Logger
data[ ].length=3 & t>3 & C_Line = prev_adc()
C_Line=data[ ].append & Timer_increment()/.
S_Line_button/Timer(Reset)
!lock_state() & direction()=true & time()=true & data[ ]=3
Secure
secure_mode()
Check Data
lock_state() & direction()=true & time()=true & data[ ]=3
direction()=false or time()=false & data[ ] !=3
!secure_mode()
S_Line_button/.
S_Line_button/.
Unsecure
time()=false
S_Line_button/.
wait()
wait()
wait()
Lock state
Unlock state
Incorrect Code
!lock_state() & time()=true
tim>2
tim=0
tim>2
tim=0
lock_state() & time()=true
tim=0
tim>2
lock
unlock
incorrect
/.lock_sound();Red_LED()
/.unlock_sound();Green_LED()
/.incorrect_sound()

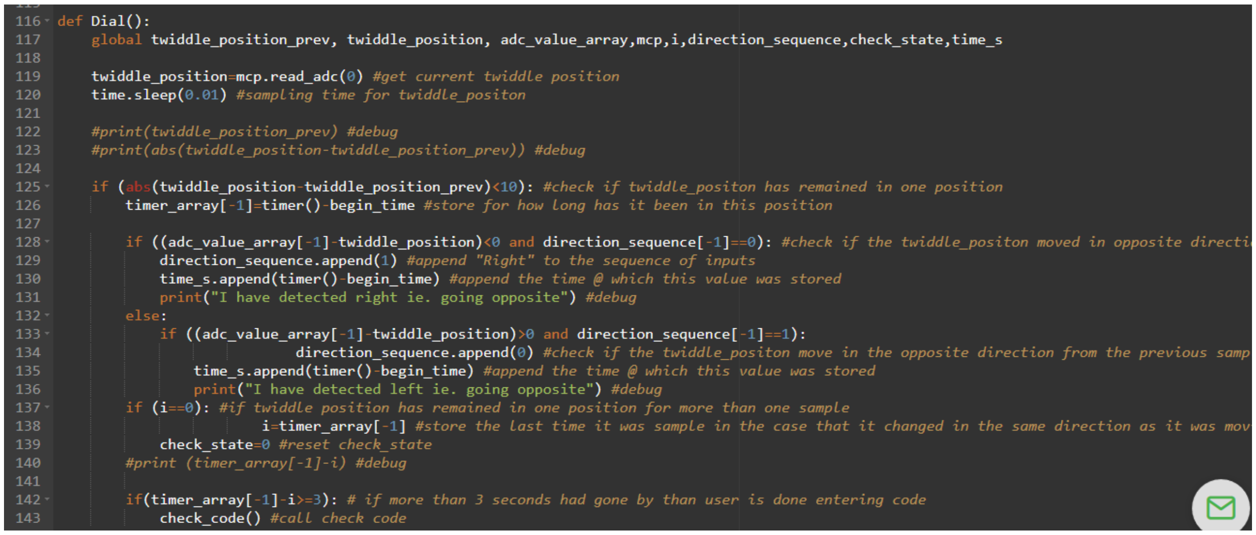
The state chart presents the operation of the whole system and the states it passes through or holds. As seen in the diagram above, the system is in idle state and is locked by default. The S_line has to be triggered to enter the code combination (transition from idle state to data logger state). This is when the values from the potentiometer (C_Knob) is read and recorded in arrays. Pressing the S_line would reset the time and direction recorded. If the value of the potentiometer does not change and the time is greater than 2 seconds, the system returns to idle state. If the correct number of arrays are filled and the time has exceeded for over 3 seconds, the system will transition to the next state where these values in arrays (time and direction) are compared. By default, the secure mode would compare both time and direction and based on the outcome, either lock, unlock or enter the incorrect data state. The unsecure mode would only compare the time and transition to the appropriate state thereafter. The lock state and unlock state would indicate a light for 2 seconds and play the appropriate sound. The incorrect code state would only play a sound. The system will then hold in this state until the S_line is pressed again to input the combinational code and repeat the process once again.



This class diagram represents the different classes the system can be associated as. This relates to the state diagram above providing the operations and attributes required in every class to transition between the different states.

## Implementation

The main code can be found in Appendix A, different functions from the code will be explained here which can be compared with the function in the state diagram provided above.



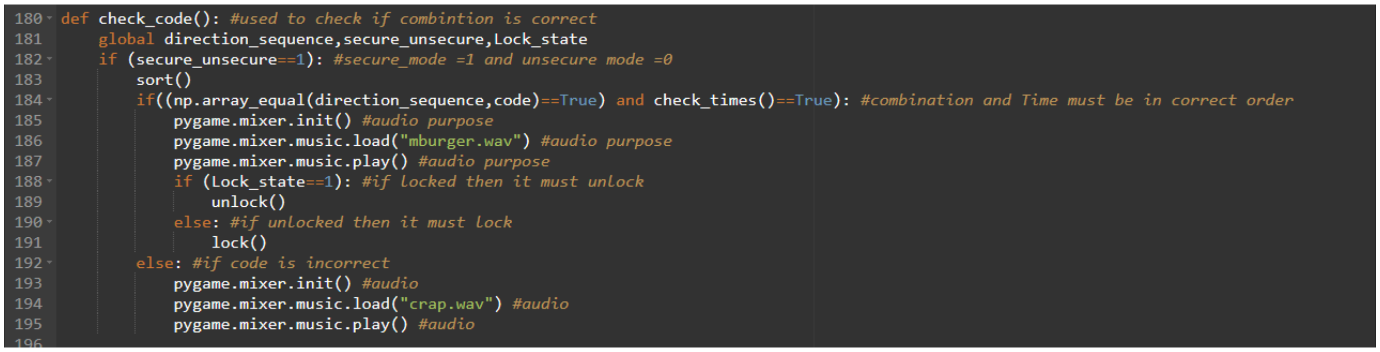
```python
def Dial():
    global twiddle_position_prev, twiddle_position, adc_value_array,mcp,i,direction_sequence,check_state,time_s

    twiddle_position=mcp.read_adc(0) #get current twiddle position
    time.sleep(0.01) #sampling time for twiddle_positon

    #print(twiddle_position_prev) #debug
    #print(abs(twiddle_position-twiddle_position_prev)) #debug

    if (abs(twiddle_position-twiddle_position_prev)<10): #check if twiddle_positon has remained in one position
        timer_array[-1]=timer()-begin_time #store for how long has it been in this position

        if ((adc_value_array[-1]-twiddle_position)<0 and direction_sequence[-1]==0): #check if the twiddle_positon moved in opposite directi
            direction_sequence.append(1) #append "Right" to the sequence of inputs
            time_s.append(timer()-begin_time) #append the time @ which this value was stored
            print("I have detected right ie. going opposite") #debug
        else:
            if ((adc_value_array[-1]-twiddle_position)>0 and direction_sequence[-1]==1):
                            direction_sequence.append(0) #check if the twiddle_positon move in the opposite direction from the previous samp
                time_s.append(timer()-begin_time) #append the time @ which this value was stored
                print("I have detected left ie. going opposite") #debug
        if (i==0): #if twiddle position has remained in one position for more than one sample
                        i=timer_array[-1] #store the last time it was sample in the case that it changed in the same direction as it was mov
            check_state=0 #reset check_state
        #print (timer_array[-1]-i) #debug

        if(timer_array[-1]-i>=3): # if more than 3 seconds had gone by than user is done entering code
            check_code() #call check code
```

The dial function determines the direction in which the dial is moving and further samples the time period in which the dial moves.

The first if statement records the time the twiddle has remained in a specific position. This is for the initial stage before the combination has been entered. This is necessary to set a new beginning time for sampling the combination.

The nested if statements determine the direction of the twiddle by comparing the initial twiddle position value and the current twiddle value position. A negative result of the difference would result in the twiddle rotating counter clockwise (left) and a positive difference would result in a clockwise rotation (right). The direction is then stored in the direction array, where left and right are represented with a 0 and 1 respectively, and the time taken for a specific rotation is stored in the timer array..

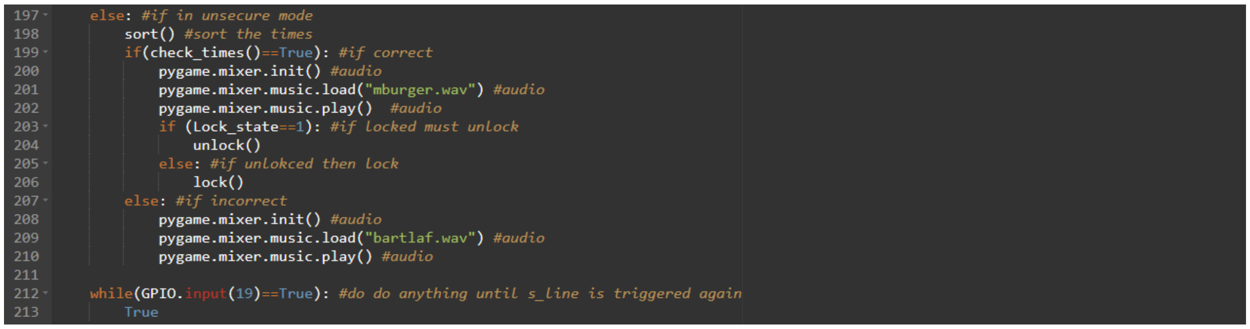
The storing of data terminates after 2 seconds and thereafter checks the combination of code.



```python
def check_code(): #used to check if combintion is correct
    global direction_sequence,secure_unsecure,Lock_state
    if (secure_unsecure==1): #secure_mode =1 and unsecure mode =0
        sort()
        if((np.array_equal(direction_sequence,code)==True) and check_times()==True): #combination and Time must be in correct order
            pygame.mixer.init() #audio purpose
            pygame.mixer.music.load("mburger.wav") #audio purpose
            pygame.mixer.music.play() #audio purpose
            if (Lock_state==1): #if locked then it must unlock
                unlock()
            else: #if unlocked then it must lock
                lock()
        else: #if code is incorrect
            pygame.mixer.init() #audio
            pygame.mixer.music.load("crap.wav") #audio
            pygame.mixer.music.play() #audio
```

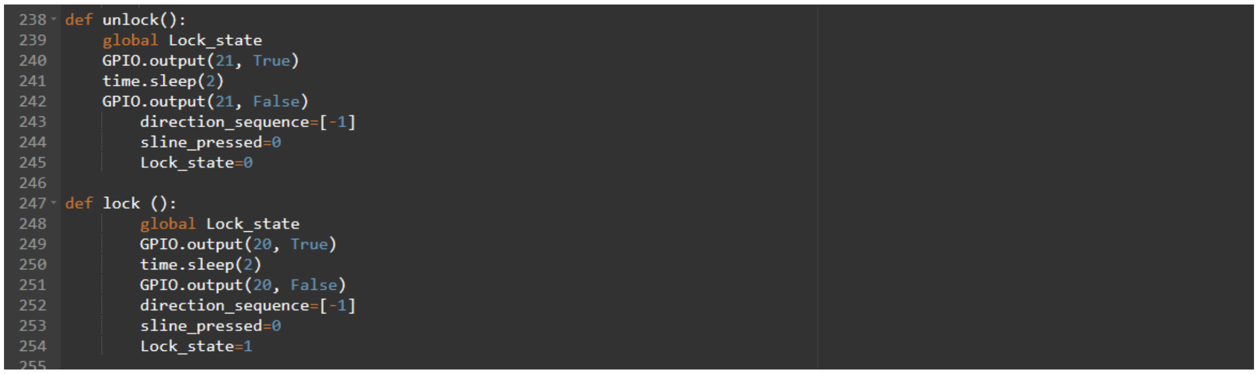
The check code function is in secure mode by default. To change to unsecure mode, requires changing the secure_unsecure variable to ‘0’.This was implement by recompiling the code and changing the variable appropriately.

The secure mode determines if the direction array matches with the code direction and further compares the time. If the correct code and time are achieved, a sound is played and if the system was locked prior it will unlock and vise versa. If the incorrect code was applied or the time did not match, a different sound would play and the lock/unlock would remain in the same state.



```
    else: #if in unsecure mode
        sort() #sort the times
        if(check_times()==True): #if correct
            pygame.mixer.init() #audio
            pygame.mixer.music.load("mburger.wav") #audio
            pygame.mixer.music.play()  #audio
            if (Lock_state==1): #if locked must unlock
                unlock()
            else: #if unlokced then lock
                lock()
        else: #if incorrect
            pygame.mixer.init() #audio
            pygame.mixer.music.load("bartlaf.wav") #audio
            pygame.mixer.music.play() #audio

    while(GPIO.input(19)==True): #do do anything until s_line is triggered again
        True
```

The unsecure mode follows the same procedure as the secure mode although only the time for the rotations are compared. In both modes, after successful or unsuccessful attempts of combination codes, the resulting state would remain in an idle state until the S_Line switch is pressed to restart the procedure. This was done to achieve a robust system.



```
def unlock():
    global Lock_state
    GPIO.output(21, True)
    time.sleep(2)
    GPIO.output(21, False)
        direction_sequence=[-1]
        sline_pressed=0
        Lock_state=0

def lock ():
        global Lock_state
        GPIO.output(20, True)
        time.sleep(2)
        GPIO.output(20, False)
        direction_sequence=[-1]
        sline_pressed=0
        Lock_state=1
```

The unlock function outputs a green light indicator for 2 seconds, resets the direction sequence and changes the variable to indicate it is in the unlock state.

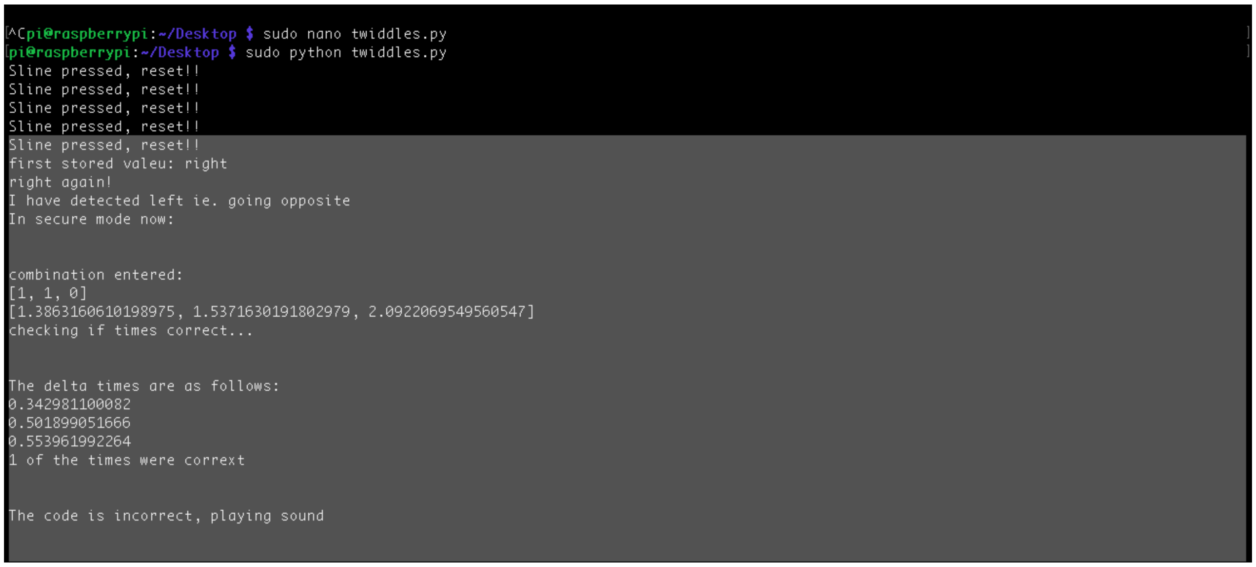
The lock function outputs a red light indicator for 2 seconds, resets the direction sequence and changes the variable to indicate it is in the lock state.

## Validation and performance

For validation and performance purposes, different testing cases were simulated where the desired output was known and was compared to the results the system produced.

### Case 1: Secure Mode – correct combination and timing is incorrect

This test was performed in secure mode where the correct combination was provided yet the timing was incorrect, the expected output should be ‘incorrect code’.

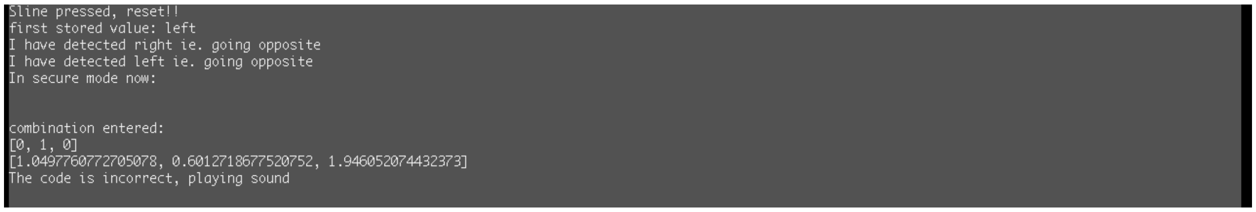


```
^Cpi@raspberrypi:~/Desktop $ sudo nano twiddles.py
pi@raspberrypi:~/Desktop $ sudo python twiddles.py
Sline pressed, reset!!
Sline pressed, reset!!
Sline pressed, reset!!
Sline pressed, reset!!
Sline pressed, reset!!
first stored valeu: right
right again!
I have detected left ie. going opposite
In secure mode now:


combination entered:
[1, 1, 0]
[1.3863160610198975, 1.5371630191802979, 2.0922069549560547]
checking if times correct...


The delta times are as follows:
0.342981100082
0.501899051666
0.553961992264
1 of the times were corrext


The code is incorrect, playing sound
```

As noticed above, only 1 of the 3 times were correct which resulted in an incorrect code. (As expected)

### Case 2: Secure mode - incorrect combination

This test was performed in secure mode where the incorrect combination was provided. The expected output should be ‘incorrect code’.

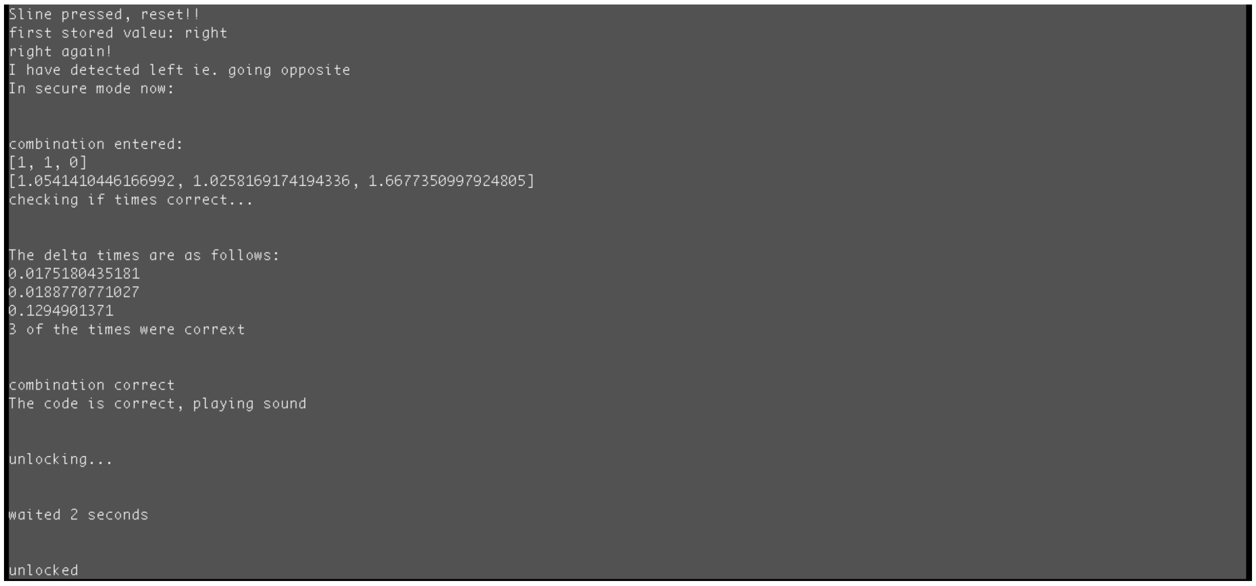


```
Sline pressed, reset!!
first stored value: left
I have detected right ie. going opposite
I have detected left ie. going opposite
In secure mode now:


combination entered:
[0, 1, 0]
[1.0497760772705078, 0.6012718677520752, 1.946052074432373]
The code is incorrect, playing sound
```

As noticed above, due to the direction of the combination code being incorrect, the timing considered, which resulted in an incorrect code (as expected). This was done so to remove redundancy for the reason that if the code is correct why should one check the timing.

### Case 3: Secure mode – correct combination and correct timing

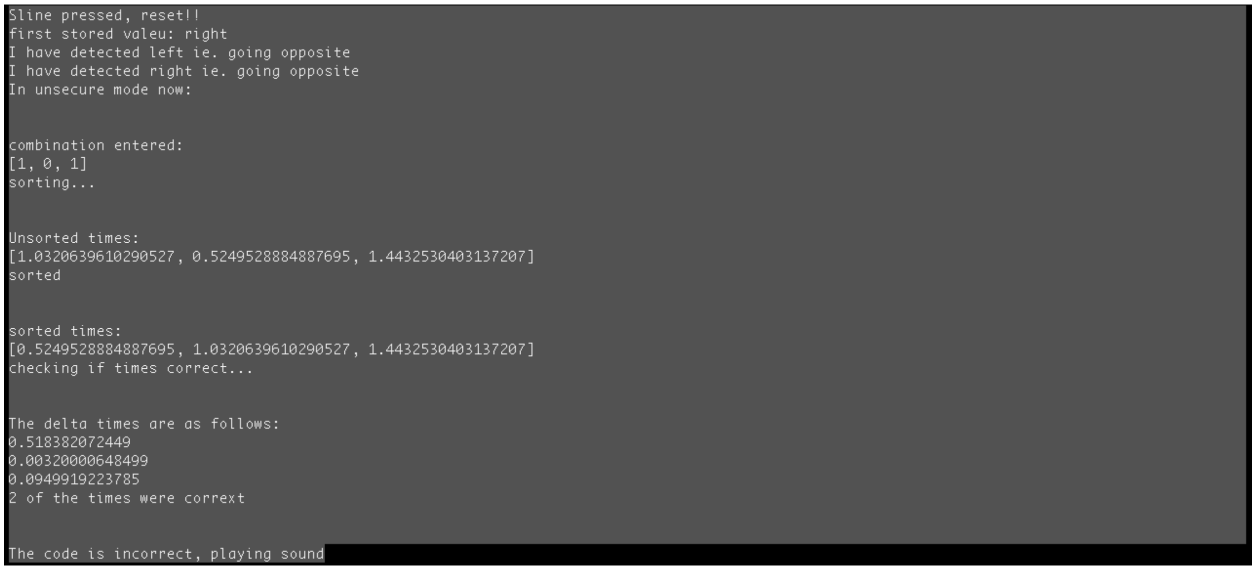
This test was performed in secure mode where the correct combination was entered and the timing was correct, the expected output should be ‘correct code’.


```
Sline pressed, reset!!
first stored valeu: right
right again!
I have detected left ie. going opposite
In secure mode now:


combination entered:
[1, 1, 0]
[1.0541410446166992, 1.0258169174194336, 1.6677350997924805]
checking if times correct...


The delta times are as follows:
0.0175180435181
0.0188770771027
0.1294901371
3 of the times were correxf


combination correct
The code is correct, playing sound


unlocking...


waited 2 seconds


unlocked
```

As noticed above, the direction of the combination code as well as the timing were both correct, resulting the system to unlock (because the system was locked in its prior state).

## Case 4: Unsecure mode – incorrect combination and incorrect timing

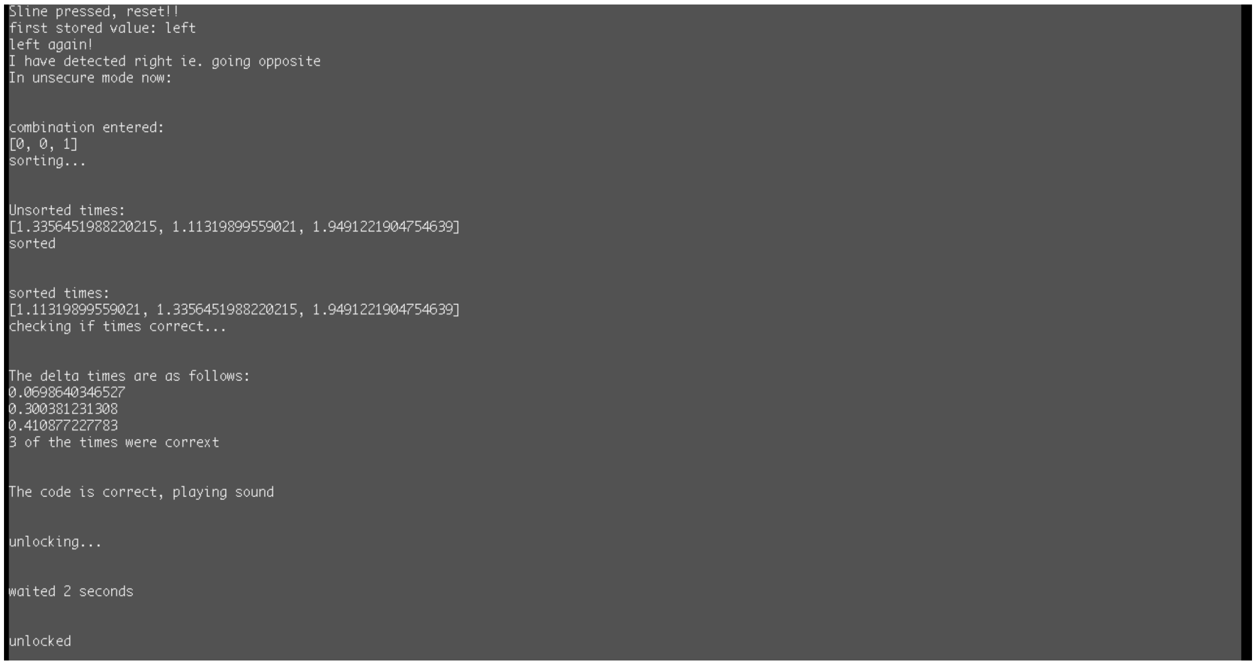
This test was performed in unsecure mode where the incorrect combination was provided and the timing was incorrect, the expected output should be 'incorrect code'. (Keep in mind that unsecure mode only evaluates the time and not the direction to lock/unlock).


```
Sline pressed, reset!!
first stored valeu: right
I have detected left ie. going opposite
I have detected right ie. going opposite
In unsecure mode now:


combination entered:
[1, 0, 1]
sorting...


Unsorted times:
[1.0320639610290527, 0.5249528884887695, 1.4432530403137207]
sorted


sorted times:
[0.5249528884887695, 1.0320639610290527, 1.4432530403137207]
checking if times correct...


The delta times are as follows:
0.518382072449
0.00320000648499
0.0949919223785
2 of the times were correxf


The code is incorrect, playing sound
```

As noticed above, only 2 of the 3 times were correct which resulted in an incorrect code. The direction is not considered in unsecure mode.

## Case 5: Unsecure mode – Incorrect combination and correct timing

This test was performed in unsecure mode where the incorrect combination was provided and the timing was correct, the expected output should be 'correct code'.



```
Sline pressed, reset!!
first stored value: left
left again!
I have detected right ie. going opposite
In unsecure mode now:


combination entered:
[0, 0, 1]
sorting...


Unsorted times:
[1.3356451988220215, 1.11319899559021, 1.9491221904754639]
sorted


sorted times:
[1.11319899559021, 1.3356451988220215, 1.9491221904754639]
checking if times correct...


The delta times are as follows:
0.0698640346527
0.300381231308
0.410877227783
3 of the times were corrext


The code is correct, playing sound


unlocking...


waited 2 seconds


unlocked
```

As noticed above, the times were correct which resulted in a correct code and unlocked the system. The direction is not considered in unsecure mode.

Based on these testing cases, we can confirm that the performance of the twiddle lock system works efficiently and consistently. Different scenarios were provided and they resulted in the correct expected output every time. The system works as designed and does provide a 2 second indicator on whether the system is locked or unlocked.

## Conclusion

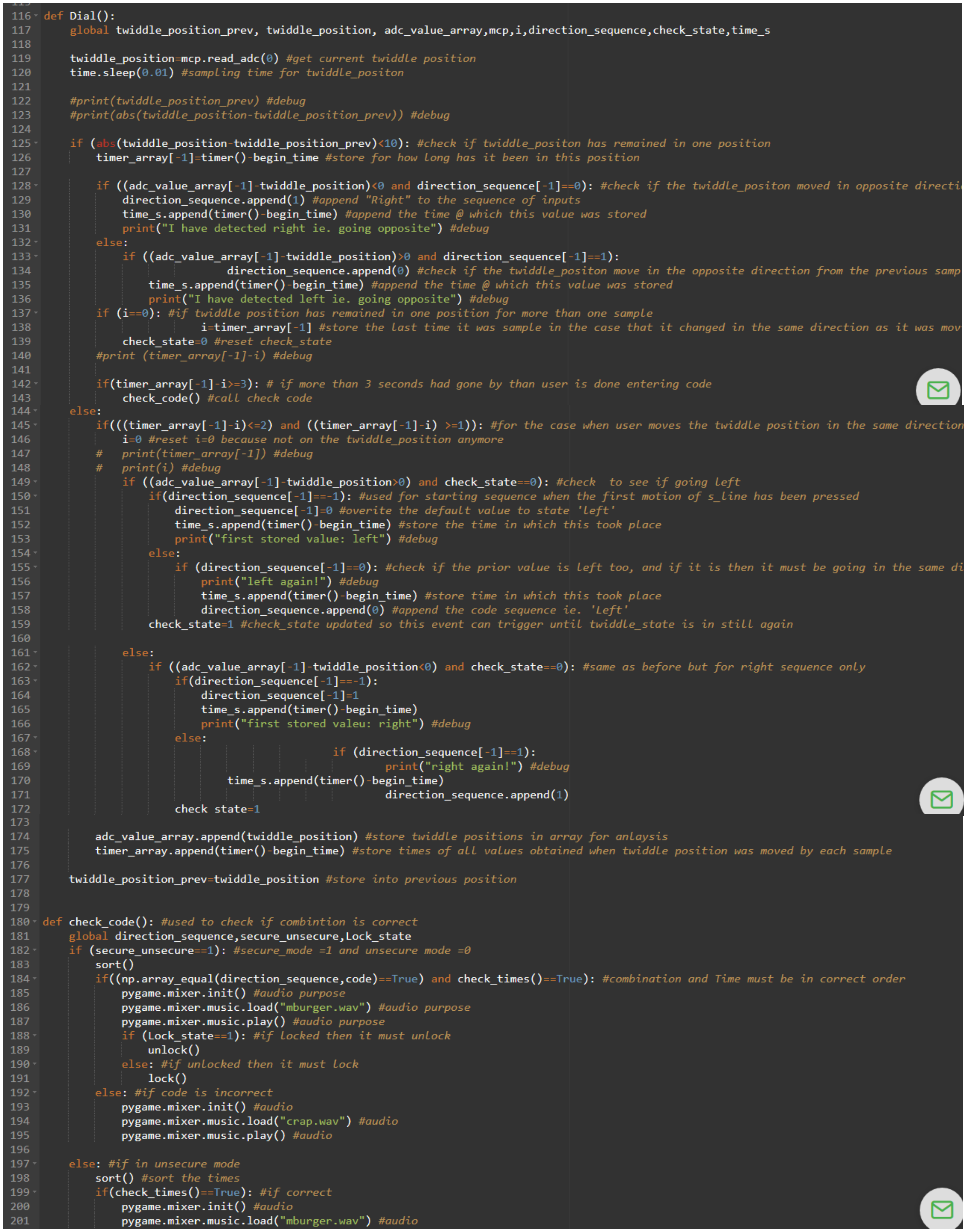
The Twiddle lock system was found to be successful as seen in the validation and performance section. The system worked efficiently and consistently which proved the design approach and implementation procedure followed in this project to be correct. Based on the testing cases simulated on the system, the results produced the desired outcome and successfully met the user requirements stated in the document above. The system designed can prove to be a very useful and suitable product which can be applied to security system as it accounts for timing as well as the direction of rotation. By implementing better potentiometers and switches, we can increase the sensitivity and reduce the uncertainty of the rotations and implement specific positions to lock or unlock the system. In final conclusion, the Twiddle lock system was successful and operated as designed for.

## References

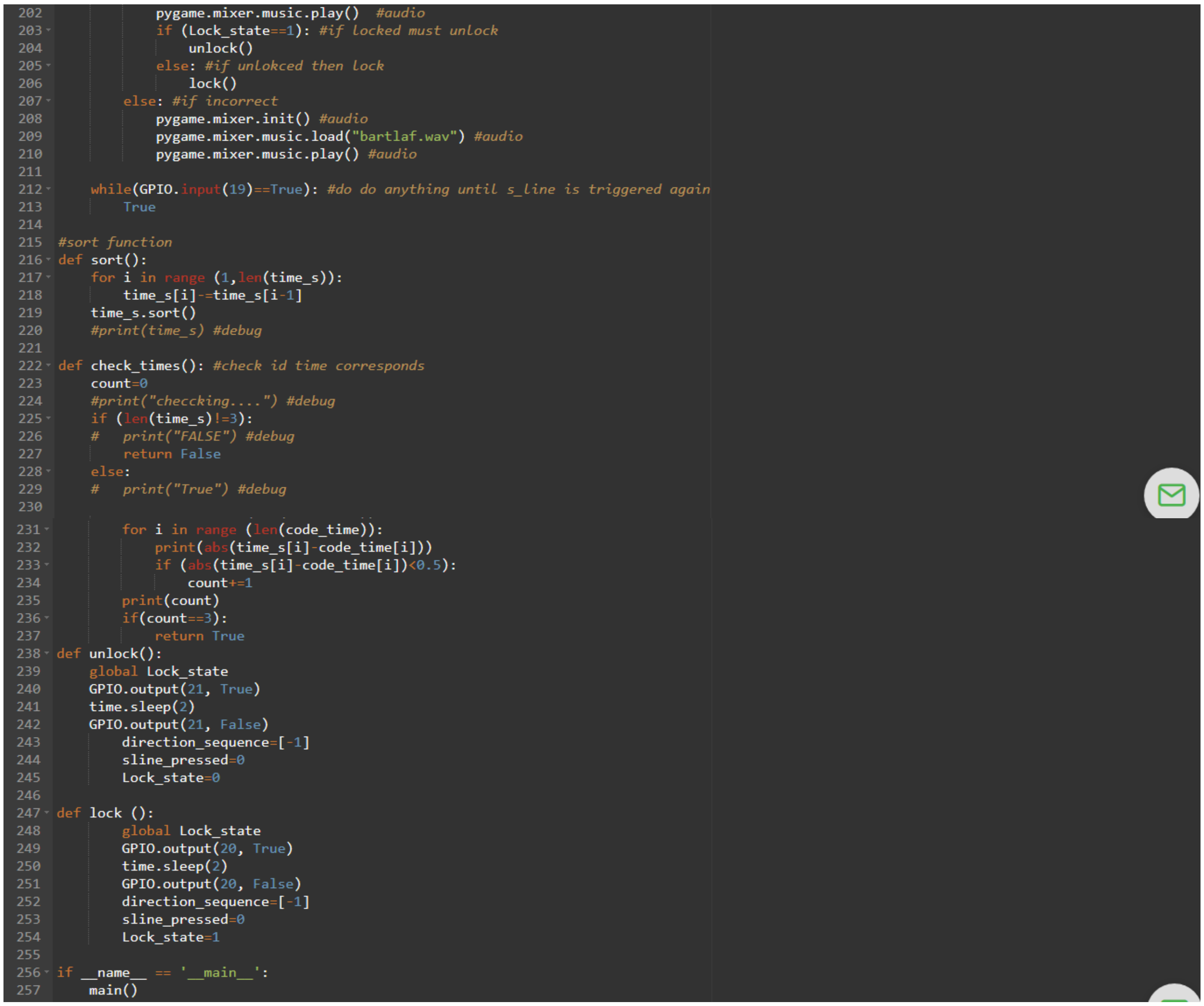
[1]C. McMurrough, "cmcmurrough/cse2312", *GitHub*, 2018. [Online]. Available: https://github.com/cmcmurrough/cse2312/blob/master/examples/array.s. [Accessed: 22- Oct- 2018].

## Appendices

### Appendix A -Code

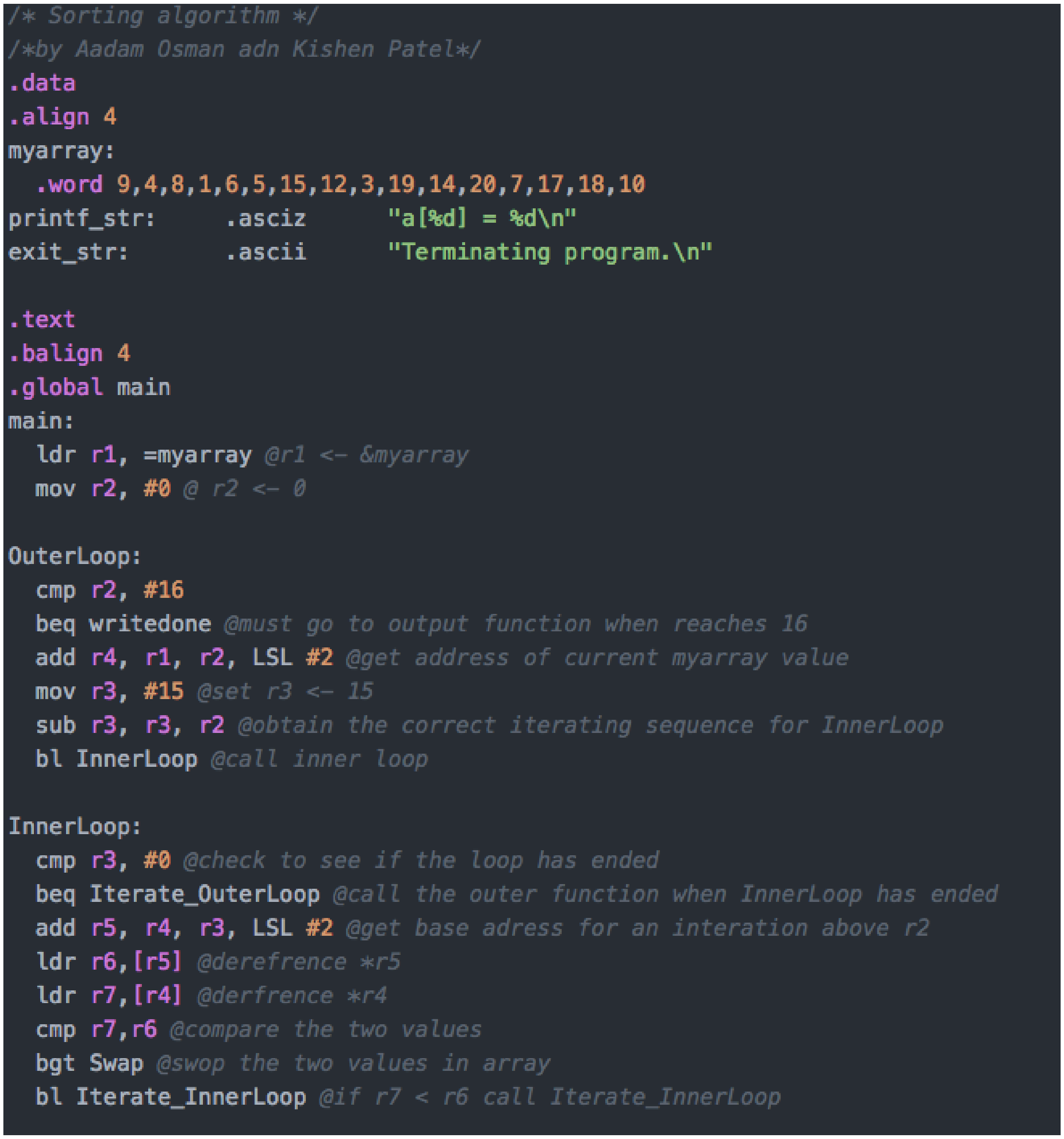


```python
def Dial():
    global twiddle_position_prev, twiddle_position, adc_value_array,mcp,i,direction_sequence,check_state,time_s

    twiddle_position=mcp.read_adc(0) #get current twiddle position
    time.sleep(0.01) #sampling time for twiddle_positon

    #print(twiddle_position_prev) #debug
    #print(abs(twiddle_position-twiddle_position_prev)) #debug

    if (abs(twiddle_position-twiddle_position_prev)<10): #check if twiddle_positon has remained in one position
        timer_array[-1]=timer()-begin_time #store for how long has it been in this position

        if ((adc_value_array[-1]-twiddle_position)<0 and direction_sequence[-1]==0): #check if the twiddle_positon moved in opposite directi
            direction_sequence.append(1) #append "Right" to the sequence of inputs
            time_s.append(timer()-begin_time) #append the time @ which this value was stored
            print("I have detected right ie. going opposite") #debug
        else:
            if ((adc_value_array[-1]-twiddle_position)>0 and direction_sequence[-1]==1):
                            direction_sequence.append(0) #check if the twiddle_positon move in the opposite direction from the previous samp
                time_s.append(timer()-begin_time) #append the time @ which this value was stored
                print("I have detected left ie. going opposite") #debug
        if (i==0): #if twiddle position has remained in one position for more than one sample
                        i=timer_array[-1] #store the last time it was sample in the case that it changed in the same direction as it was mov
            check_state=0 #reset check_state
        #print (timer_array[-1]-i) #debug

        if(timer_array[-1]-i>=3): # if more than 3 seconds had gone by than user is done entering code
            check_code() #call check code
    else:
        if(((timer_array[-1]-i)<=2) and ((timer_array[-1]-i) >=1)): #for the case when user moves the twiddle position in the same direction
            i=0 #reset i=0 because not on the twiddle_position anymore
        #   print(timer_array[-1]) #debug
        #   print(i) #debug
            if ((adc_value_array[-1]-twiddle_position>0) and check_state==0): #check  to see if going left
                if(direction_sequence[-1]==-1): #used for starting sequence when the first motion of s_line has been pressed
                    direction_sequence[-1]=0 #overite the default value to state 'left'
                    time_s.append(timer()-begin_time) #store the time in which this took place
                    print("first stored value: left") #debug
                else:
                    if (direction_sequence[-1]==0): #check if the prior value is left too, and if it is then it must be going in the same di
                        print("left again!") #debug
                        time_s.append(timer()-begin_time) #store time in which this took place
                        direction_sequence.append(0) #append the code sequence ie. 'Left'
                check_state=1 #check_state updated so this event can trigger until twiddle_state is in still again

            else:
                if ((adc_value_array[-1]-twiddle_position<0) and check_state==0): #same as before but for right sequence only
                    if(direction_sequence[-1]==-1):
                        direction_sequence[-1]=1
                        time_s.append(timer()-begin_time)
                        print("first stored valeu: right") #debug
                    else:
                                            if (direction_sequence[-1]==1):
                                                    print("right again!") #debug
                            time_s.append(timer()-begin_time)
                                                    direction_sequence.append(1)
                    check_state=1

            adc_value_array.append(twiddle_position) #store twiddle positions in array for anlaysis
            timer_array.append(timer()-begin_time) #store times of all values obtained when twiddle position was moved by each sample

    twiddle_position_prev=twiddle_position #store into previous position


def check_code(): #used to check if combintion is correct
    global direction_sequence,secure_unsecure,Lock_state
    if (secure_unsecure==1): #secure_mode =1 and unsecure mode =0
        sort()
        if((np.array_equal(direction_sequence,code)==True) and check_times()==True): #combination and Time must be in correct order
            pygame.mixer.init() #audio purpose
            pygame.mixer.music.load("mburger.wav") #audio purpose
            pygame.mixer.music.play() #audio purpose
            if (Lock_state==1): #if locked then it must unlock
                unlock()
            else: #if unlocked then it must lock
                lock()
        else: #if code is incorrect
            pygame.mixer.init() #audio
            pygame.mixer.music.load("crap.wav") #audio
            pygame.mixer.music.play() #audio

    else: #if in unsecure mode
        sort() #sort the times
        if(check_times()==True): #if correct
            pygame.mixer.init() #audio
            pygame.mixer.music.load("mburger.wav") #audio
```

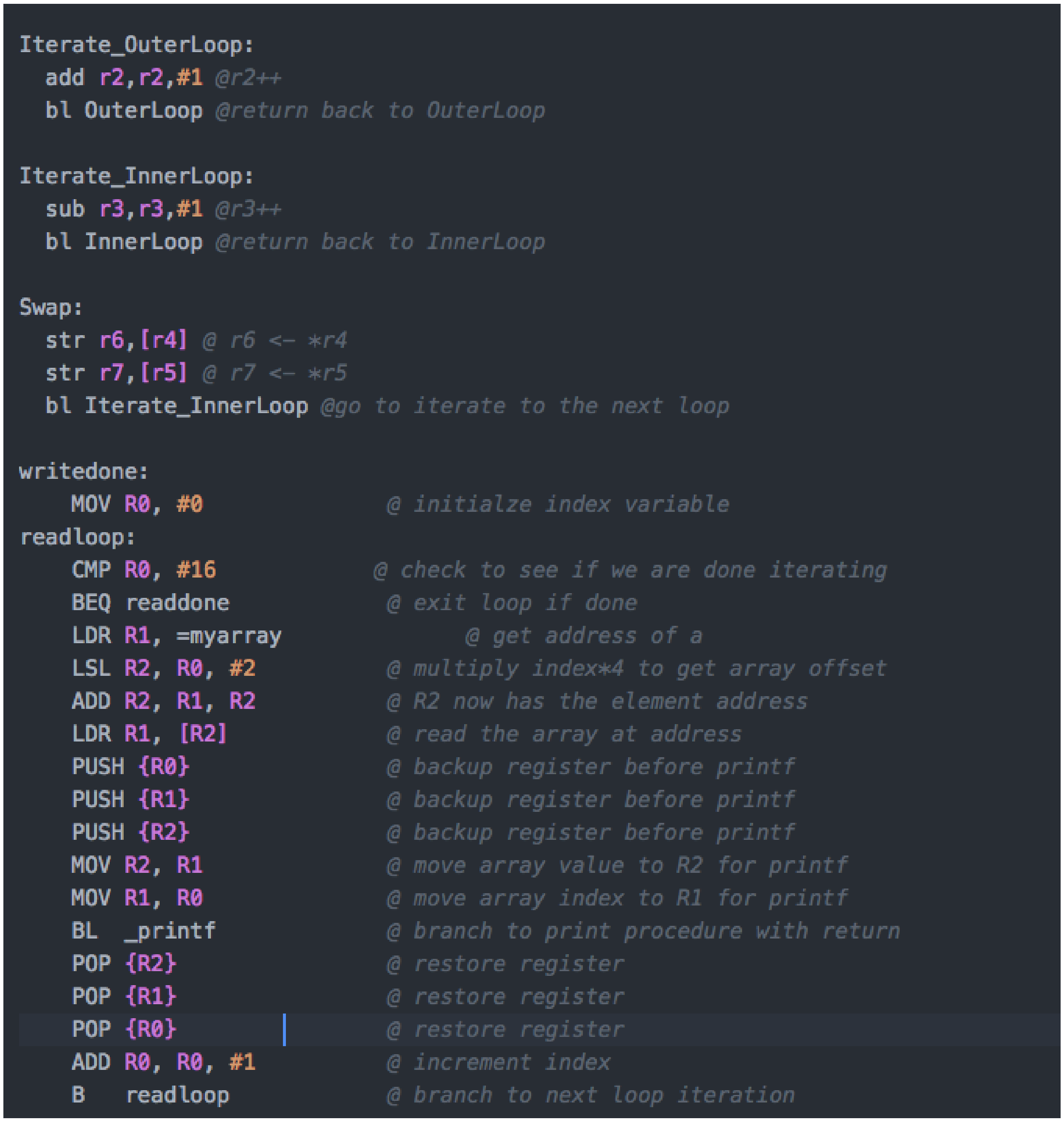


```python
            pygame.mixer.music.play()  #audio
            if (Lock_state==1): #if locked must unlock
                unlock()
            else: #if unlokced then lock
                lock()
        else: #if incorrect
            pygame.mixer.init() #audio
            pygame.mixer.music.load("bartlaf.wav") #audio
            pygame.mixer.music.play() #audio

    while(GPIO.input(19)==True): #do do anything until s_line is triggered again
        True

#sort function
def sort():
    for i in range (1,len(time_s)):
        time_s[i]-=time_s[i-1]
    time_s.sort()
    #print(time_s) #debug

def check_times(): #check id time corresponds
    count=0
    #print("checcking....") #debug
    if (len(time_s)!=3):
    #   print("FALSE") #debug
        return False
    else:
    #   print("True") #debug

        for i in range (len(code_time)):
            print(abs(time_s[i]-code_time[i]))
            if (abs(time_s[i]-code_time[i])<0.5):
                count+=1
        print(count)
        if(count==3):
            return True
def unlock():
    global Lock_state
    GPIO.output(21, True)
    time.sleep(2)
    GPIO.output(21, False)
        direction_sequence=[-1]
        sline_pressed=0
        Lock_state=0

def lock ():
        global Lock_state
        GPIO.output(20, True)
        time.sleep(2)
        GPIO.output(20, False)
        direction_sequence=[-1]
        sline_pressed=0
        Lock_state=1

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

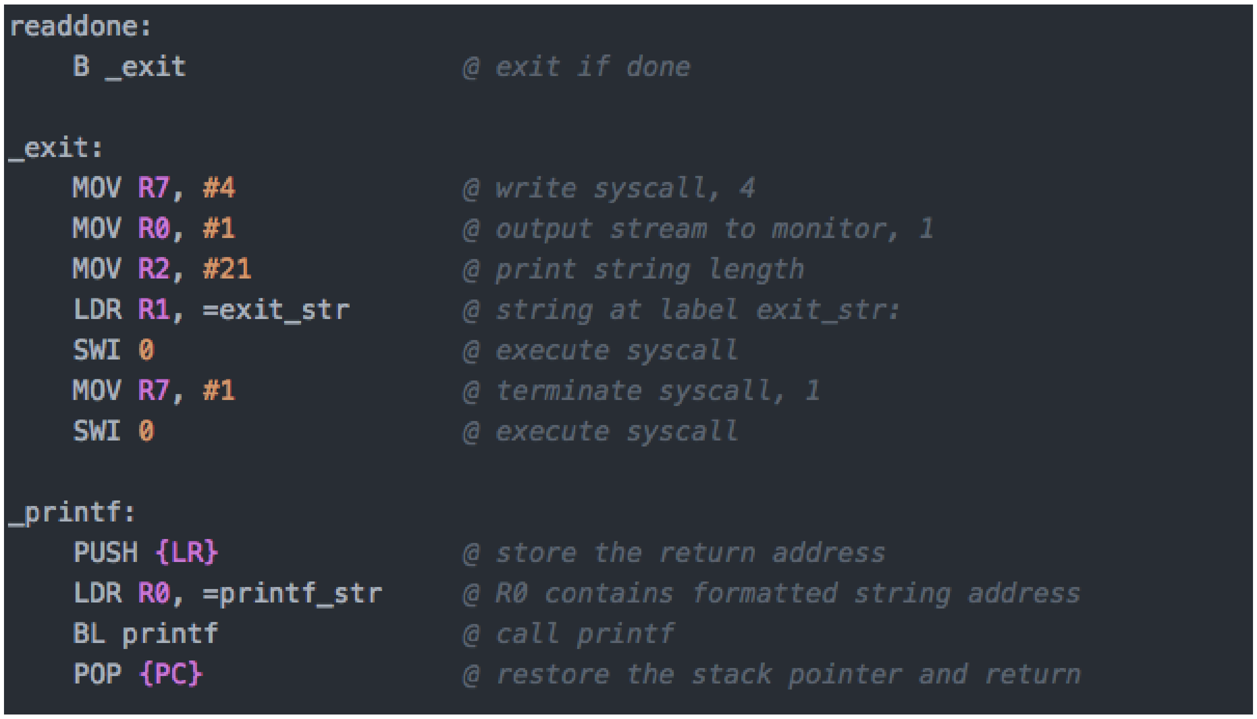
## Appendix B: Assembly code for Sorting function



```
/* Sorting algorithm */
/*by Aadam Osman adn Kishen Patel*/
.data
.align 4
myarray:
  .word 9,4,8,1,6,5,15,12,3,19,14,20,7,17,18,10
printf_str:     .asciz      "a[%d] = %d\n"
exit_str:       .ascii      "Terminating program.\n"

.text
.balign 4
.global main
main:
  ldr r1, =myarray @r1 <- &myarray
  mov r2, #0 @ r2 <- 0

OuterLoop:
  cmp r2, #16
  beq writedone @must go to output function when reaches 16
  add r4, r1, r2, LSL #2 @get address of current myarray value
  mov r3, #15 @set r3 <- 15
  sub r3, r3, r2 @obtain the correct iterating sequence for InnerLoop
  bl InnerLoop @call inner loop

InnerLoop:
  cmp r3, #0 @check to see if the loop has ended
  beq Iterate_OuterLoop @call the outer function when InnerLoop has ended
  add r5, r4, r3, LSL #2 @get base adress for an interation above r2
  ldr r6,[r5] @derefrence *r5
  ldr r7,[r4] @derfrence *r4
  cmp r7,r6 @compare the two values
  bgt Swap @swop the two values in array
  bl Iterate_InnerLoop @if r7 < r6 call Iterate_InnerLoop
```



```
Iterate_OuterLoop:
  add r2,r2,#1 @r2++
  bl OuterLoop @return back to OuterLoop

Iterate_InnerLoop:
  sub r3,r3,#1 @r3++
  bl InnerLoop @return back to InnerLoop

Swap:
  str r6,[r4] @ r6 <- *r4
  str r7,[r5] @ r7 <- *r5
  bl Iterate_InnerLoop @go to iterate to the next loop

writedone:
    MOV R0, #0              @ initialze index variable
readloop:
    CMP R0, #16            @ check to see if we are done iterating
    BEQ readdone            @ exit loop if done
    LDR R1, =myarray              @ get address of a
    LSL R2, R0, #2          @ multiply index*4 to get array offset
    ADD R2, R1, R2          @ R2 now has the element address
    LDR R1, [R2]            @ read the array at address
    PUSH {R0}               @ backup register before printf
    PUSH {R1}               @ backup register before printf
    PUSH {R2}               @ backup register before printf
    MOV R2, R1              @ move array value to R2 for printf
    MOV R1, R0              @ move array index to R1 for printf
    BL  _printf             @ branch to print procedure with return
    POP {R2}                @ restore register
    POP {R1}                @ restore register
    POP {R0}                @ restore register
    ADD R0, R0, #1          @ increment index
    B   readloop            @ branch to next loop iteration
```



```
readdone:
    B _exit                  @ exit if done

_exit:
    MOV R7, #4               @ write syscall, 4
    MOV R0, #1               @ output stream to monitor, 1
    MOV R2, #21              @ print string length
    LDR R1, =exit_str        @ string at label exit_str:
    SWI 0                    @ execute syscall
    MOV R7, #1               @ terminate syscall, 1
    SWI 0                    @ execute syscall

_printf:
    PUSH {LR}                @ store the return address
    LDR R0, =printf_str      @ R0 contains formatted string address
    BL printf                @ call printf
    POP {PC}                 @ restore the stack pointer and return
```

The code above, in ARM assembly, follows the procedure of iterating through each element in 'myarray' and compares itself to each individual element value, and if the current value is found to be greater than the current value it will 'Swop' the two values in the array. These iterations are successfully completed using the OuterLoop and InnerLoop functions, along with r1 to r7 registers.

The means in which the sorted array was printed was done so using the functions adapted from an open source code found of GitHub by C.McMurrough. [1]