Prac 6: Twiddle Lock Project

David Fransch[†] and Richard Powrie[‡]
EEE3096S 2018
University of Cape Town
South Africa
[†]FRNDAV011 [‡]PWRRIC001

***Abstract*—The following report details the development of the Twiddle Lock, an electronic version of the common safe dial lock. The report describes the requirements, modelling, implementation and performance evaluation of the system.**

I. Introduction

The following report describes the design and implementation of the Twiddle Lock for the final Embedded Systems practical.

The Twiddle lock is comparable to an electronic version of the classical dial combination safe, lock mechanism.

The project made use of the Raspeberry Pi 3B (RPi) and was developed using Python code.

The application works by reading in the input of a button which indicates that the user is about to put in a 'dialed code'. The dialed code is sensed by an ADC connected to a potentiometer (POT) acting as the dial.

The RPi takes in two GPIO inputs namely the S-line (pushbutton) and C-line (ADC connected to POT).

The application has two GPIO output lines to signify to the user of the locked state (i.e. locked or unlocked). The output lines are the L-line and U-Line for locked and unlocked respectively.

The lock works in two modes namely, secure and unsecure.

In secure mode the system checks for the direction of turning as well as the duration for which a turn takes place and is concerned with the order of the entered combination.

Unsecure mode simply considers the duration of turns and does not consider the order in which they are entered.

The system block diagram is shown in Figure 1.

II. Requirements

User Requirements

1) The user should be able to operate the lock in either secure or unsecure mode.
2) The user must press the s-button before entering in a code.

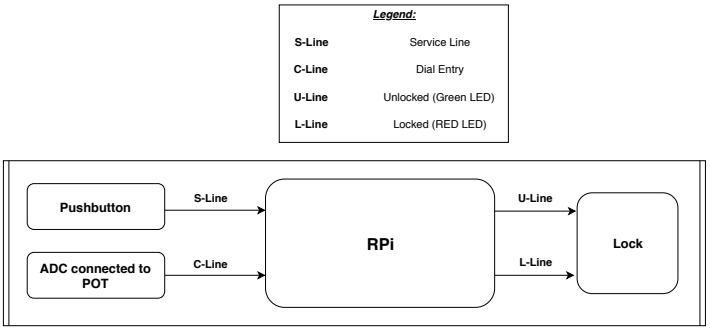

Fig. 1. System Block Diagram

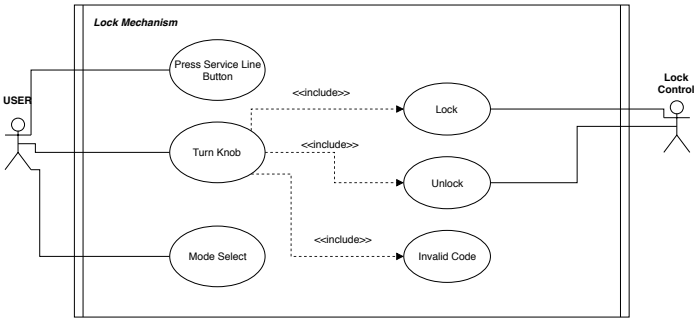

Fig. 2. Use Case Diagram

3) After pressing s-button the user will have a limited time to enter in the password/code.
4) User is able to enter directions and their respective durations by turning knob on POT.
5) An additional feature is that the user will be notified by sound of a successful code entry and unsuccessful code entry.

Use Case Description

The Use Case diagram in Figure 2 describes the basic requirements of the system.

In this diagram there are two actors namely, the user and the lock control.

The user can select mode (or use default secure mode), press the s-line button before entering in the code and finally enter in the code by turning the knob attached to the POT.

Depending on the initial lock state, the lock control will either lock or unlock for a valid code or provide the incorrect code display to the user.

III. Specifications and Design

Software Specifications

1) Main function that performs startup, configuration and controls input/output lines.
2) Store a log of durations and directions in two arrays respectively.
3) Develop a sort rotuine for unsecure mode (Python). Similiarly develop a sort routine in ARM assembly.
4) Hard code a lock/unlock sequence which will be matched with the user input.
5) Adjust timing to make application easier to use.

State Chart Description

The activity diagram shown in Figure 3 describes the main operation of the Twiddle Lock system.

The user intially sets the mode to be either secure or unsecure.

The user must then press the S-button to reset the system and prepare the system to allow entering of the code.

Pressing the S-button provides the user with console message describing the selected mode and prompting the user to enter the code.

Once the S-button is pressed the user can then enter in the code using the POT (labelled C-Adjust). This sends the action to the ADC which is decoded within the lock mechanism section.

The lock mechanism section, checks for the correct duration for unsecure mode and checks both the direction and duration of turns for the secure mode.

Depending on the lock state (i.e. locked or unlocked) the correct code will change state from locked to unlocked or vice versa.

If the state was originally locked and the correct code is entered the lock will unlock.

This is shown to the user by a flash of the green LED (unlocked LED) as well as user message and sound notifying the user of the unlocked state.

If the state was originally unlocked the correct code will similiarly notify the user with the flash of the red LED (locked LED) as well as user message which specifies the newly locked state and success sound.

If the wrong code is entered the red LED (locked LED) will aslo flash and a message and unsuccess sound notifying the user of an incorrect code will be displayed.

Sequence Diagram Description

The sequence diagram in Figure 4 consists of four actors namely; the User, S-button, RPi (where the main code is), C-POT to ADC and the User Output along with their respective lifelines.

The user intially sets the mode and presses the S-button to start the process of entering in the code. These are shown as asynchronous messages.

The pressing of the S-button notifies the User Output which responds with an asynchronous reply message, prompting the user to enter in the code. The timer is also reset which is shown as a synchronous message since this happens across the system.

The next action is labelled C-Adjusted and this refers to the user adjusting the POT to enter in the code.

A synchronous reply message is sent from the ADC to the RPi containing the entered duration and directions.

The RPi code evaluates the mode, direction and duration and sends a synchronous message to the User Output.

Depending on the correctness of the code, the User Output responds accordingly.

IV. Implementation

System Implementation - Python Code

This section provides code snippets with their respective descriptions.

The function of the S-line which resets the system and prompts the user to enter the code is shown in Listing 1. This is implemented using an interrupt which always checks when the button is pressed.

This function resets the necessary system timers and intialises the entered variable to be true, which starts the main function of the program.

The S-button also provides user output to notify the user of the mode they are in.

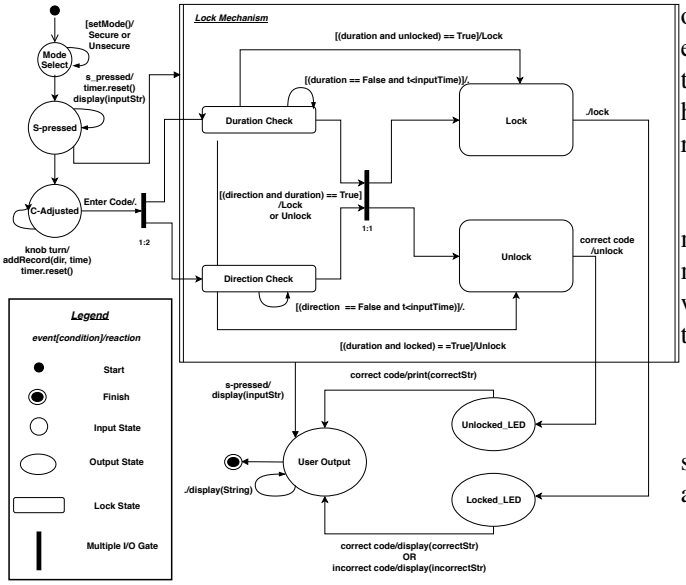

Fig. 3. System State Chart

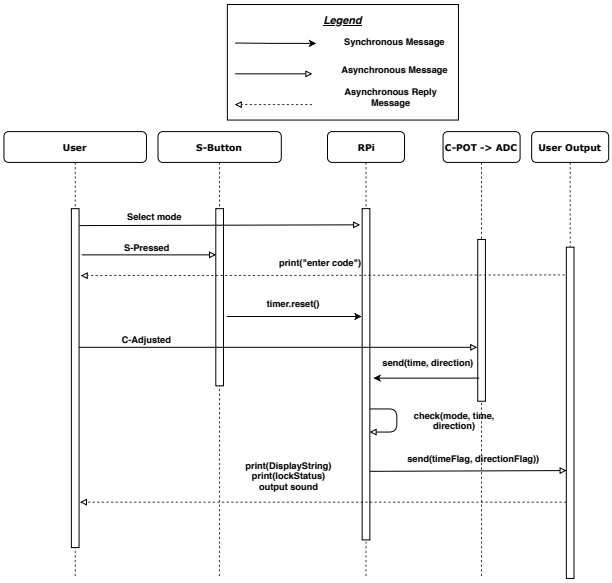

Fig. 4. System Sequence Diagram

In Listing 2 the code describes how the direction and duration of turns are recorded within the main while loop of the program. The variable epsilom is used as a fixed tolerance associated with the turning of the POT, this was set to 0.1.

The Add Record function seen in Listing 3 shows how the program stores the duration and direction. The respective values are stored in an array using a first in last out (FILO) method.

The secure mode implementation is shown in Listing 4, in this mode both direction and duration entered in the correct order are required for the code entry. The first if statement ensures that the user is in secure mode and they are within the input time limit. The code entry is then compared to the hardcoded correct code and depending on correctness, the respective LED and user message is displayed to the user.

Finally the unsecure code is shown in Listing 5. In this mode only the correct duration is required. When secure mode is set to false unsecure mode checks that the user is within the input time evaluates the code entry and provides the required output display for the user.

Assembly Implementation

A sort function was required to be written in assembly, the sort method used was comparison sort. The assembly code is attached at the end of this report in Appendix A.

V. Validation and Performance

System Performance

- Voltage tolerance:
  This tolerance is stored in the varaible epsilom and represents the allowable change in voltage without changing direction. Epsilom is set to 0.01.

- Allowed pause time:
  This is the allowed pause between entries and is set to 2 seconds to give the user enough time to change direction and distinguish between single entries.
- Input time:
  The input time is the amount of time the user has to enter in the required code. This value was set to 6 seconds.

In Table II averages of the pause time and input time are compared. The first time is taken from python timing code (time.time()) and the second is recorded using a stopwatch.

The times are quite close and the disparity between times could be a result of delay in pressing the start and stop button on the stopwatch.

All values were intially set using reasonable intuition - as testing proceeded timing specifications were adjusted to accomodate for a more natural user experience.

This was further refined by testing the system with multiple users to get valid feedback.

System Testing

The first mode that is tested is secure mode, in this mode both the direction and duration of entries must match the hardcoded code.

In Figure 5 the state of the lock is initially locked (default state), when the correct code is entered the figure shows that the lock becomes unlocked. This is displayed to the user in

```
def callback1(button1):#reset
    global timerStart
    global displayOn
    global entered
    global secureTimerStart
    global attempts

    attempts = 0
    secureTimerStart = time.time()
    entered = True
    timerStart = time.time()#reset timer
    os.system("clear")
    if (displayOn==True):
        print("Reset pressed")
        print("Enter combination")

    if(secureMode == True):
        print("entered secure mode")
    else:
        print("unsecure mode")
    clearArr()
```

Listing 1. S-Line implementation.

```
            if(currentDir == left and currentVal>=(prevVal+epsilom)):

                if(pauseTimer<allowedPause):
                    addRecord(currentDir,generalTimer)#add record of direction and time

                generalTimerStart = time.time()#reset timer
                currentDir = right

            elif(currentDir == right and currentVal<=(prevVal-epsilom)):

                if(pauseTimer<allowedPause):
                    addRecord(currentDir,generalTimer)#add record of direction and time

                generalTimerStart = time.time()#reset timer
                currentDir = left
```

Listing 2. Recording of turned direction and respective duration.

```
#add record
def addRecord(direction, timevalue):
    global dir
    global log
    global attempts

    for i in range(length-1,-1,-1):#shifts everything in array up by one
        if(i==0):
            dir[i] = direction
            log[i] = int(round(timevalue))
        else:
            dir[i] = dir[i-1]
            log[i] = log[i-1]

    attempts = attempts +1
    print("\n---------Entry", attempts ,"----------")
    print("\ndirection: ",dir[0])
    print("time:      ", log[0])
    print("\n-------------------------\n")
```

Listing 3. Add Record function

```
   #secure mode
    if(secureMode == True and secureTimer > inputTime):
           count = 0
           for i in range (0, 3):

                if((dir[i] == combDirection[i]) and (log[i] == combTime[i])):
                    count = count +1

            if(count == 3 and locked):
                print("correct combo, now unlocked")
                locked = False
                U_lineOut()
            elif(count ==3 and not locked):
                print("correct combo, now locked")
                locked = True
                 L_lineOut()
             else:
                print("wrong")

            clearArr()
            entered = False
```

Listing 4. Secure mode code

the text shown below the duration and direction arrays as well as by flashing the green LED for 2 seconds (i.e. unlock LED).

Figure 6 shows that when the correct code is entered and

```
#unsecure mode
elif(secureTimer > inputTime):

    print("Time", log[0:3])
    arrSort(log[0:3])
    arrSort(combinationTime)

    count = 0
    for i in range (0, 3):
        if(log[i] == combinationTime[i]):
            count = count +1
    if(count == 3 and locked):
        print("\ncorrect combo, now unlocked")
        locked = False

        U_lineOut()

    elif(count ==3 and not locked):
        print("\ncorrect combo, now locked")
        locked = True
        L_lineOut()
    else:
        print("wrong")
    clearArr()
    entered = False
```

Listing 5. Unsecure mode code

TABLE I

HARDWARE DESCRIPTION OF LAPTOP USED FOR PRACTICAL

Description	Details
Processor	Intel Core i5-7200U @ 2.5Ghz
System Type	x64
Installed RAM	12GB
Operating Sytem	Windows 10

TABLE II

TIMING EVALUATION

Description	Python Timer (s)	Recorded Time (s)
Pause Time	1.005	2.01
Input Time	6.528	7.11

the state of the lock is unlocked. The lock will become locked.

The second mode is unsecure mode and this only requires that the duration be correct.

In Figure 7 the state of lock is initially locked and becomes unlock upon the user entering the correct code, similiarly in Figure 8 the state changes to locked.

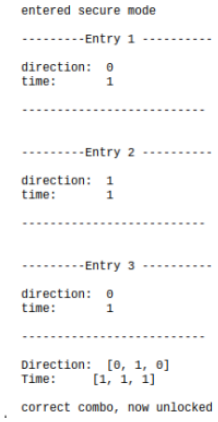

Fig. 5. Secure mode: unlock test

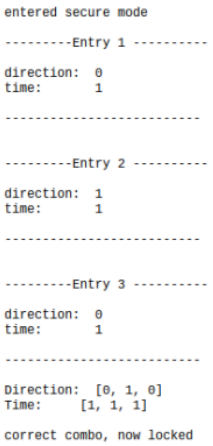

Fig. 6. Secure mode: lock test

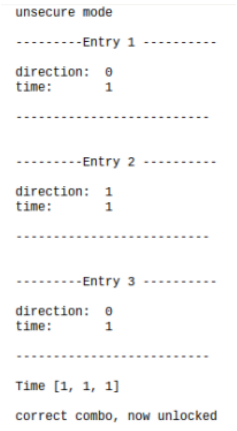

Fig. 7. Unsecure mode: unlock test

VI. Conclusion

This report detailed the process involved in developing a Twiddle Lock.

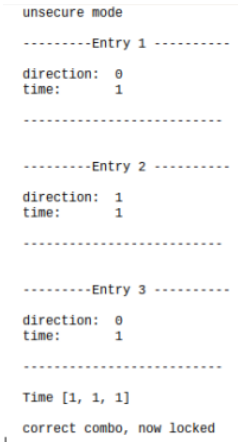

Fig. 8. Unsecure mode: lock test

The system was tested and met the requirements and specifications intially set out. More specifically a user is able to unlock or lock the system by entering in a correct code.

The code is entered in by turning a potentiometer which is connected to the ADC. The directions and durations of turns were stored in two arrays and these arrays could be checked to see if the entered code matched the required hardcoded code.

Initially arbitrary timing values and voltage tolerances were used, this made the system difficult to use. Through experimental testing, the values were refined to give the system a more natural and user friendly feel (making it easier to enter in values).

Finally, the system was tested by Simon Winberg (course lecturer) who was kind enough to mark our demo. His approval of the system performance signified that the system was indeed working correctly.

The working system is an interesting model that could be used for gaming applications where a user is required to make turns and the system must respond to these turns.

A possibly more useful implementation could be a turn tracking device which could measure anything from turbine revolutions to vehicle torque (derived from the revelotions per minute).

As it stands the lock system is probably not the most effective for its current purpose, as security measures such as biometrics are more user friendly and fail safe.

References

EEE3096S Course Notes (2018)

VII. Appendix A

```
/*-------------------------------------------------------------*/
/*EEE3096S_Lab6_assembly.s                                     */
/*Authors: Richard Powrie (PWRRIC001)                          */
/*         David Fransch (FRNDAV011)                           */
/* to run:(in command line)                                    */
/* make                                                        */
/* ./assem ; echo $?                                           */
/*-------------------------------------------------------------*/
@.func main      /*'main' is a function*/

/*-------------------------------------------------------------*/
/* --data section*/
/*-------------------------------------------------------------*/
.data

/* --arrays*/
my_array: .word 82, 70, 93, 77, 6, 23, 99, 70, 25, 30
.balign 4
len_my_array: .word 10      /*length of array = 10*/

/* --output strings*/
unsorted_message:      .asciz "The unsorted array is:\n"
sorted_message:  .asciz "The sorted array is:\n"
debug:                 .asciz "debug: %d  "

/* --format strings for array print function*/
integer_printf:  .asciz "%d "     /*print integer with space*/
newline_print:         .asciz "\n" /*print newline*/

/*-------------------------------------------------------------*/
/* --code section*/
/*-------------------------------------------------------------*/
.text



/*--------------------print_array function----------------------*/
print_array: /*paramters: r0 = address of integer array, r1 = num of
elements */
     push {r4, r5, r6, lr} /*keep lr in the stack*/
/*-------------------------------------------------------------*/
     mov r4, r0             /*store address of array*/
     mov r5, r1             /*store number of elements*/
     mov r6, #0             /*counter: current print element*/

     /*---------while loop----------*/
     Lprint_array_condition:
     cmp r6, r5 /*check condition, loop while r6 != r5*/
     beq Lprint_array_end   /*if condition met, exit loop*/
          /*prepare call to printf*/
          ldr r0, =integer_printf
          ldr r1, [r4, r6, LSL #2]     /*r1 is address: r4+r6*4*/
          bl printf
```

```
            add r6, r6, #1              /*increment r6*/
            b Lprint_array_condition    /*loop back*/

      Lprint_array_end:
      /*-----------------------------*/

      /*prepare call to puts*/
      ldr r0, =newline_print
      bl puts                           /*print new line*/

/*-------------------------------------------------------------*/
      pop {r4, r5, r6, lr}   /*restore lr from stack*/
      bx lr       /*exit program*/
/*-----------------exit print_array function--------------------*/




/*--------------------sort_array function-----------------------*/
sort_array: /*paramters: r0 = address of integer array, r1 = num of
elements */
      push {r4, r5, r6, lr} /*keep lr in the stack*/
/*-------------------------------------------------------------*/
      mov r4, r0             /*store address of array*/
      mov r5, r1             /*store number of elements*/
      sub r5, r5, #1              /*subtract 1 from length to keep loops
in bounds*/

      mov r6, #0             /*counter1: outer loop counter*/
      /*---------while loop 1--------*/
      Lsort_array_condition1:
      cmp r6, r5 /*check condition, loop while r6 != r5*/
      beq Lsort_array_end1   /*if condition met, exit loop*/


            mov r7, #0             /*counter2: inner loop counter*/
            /*---------while loop 2--------*/
            Lsort_array_condition2:
            cmp r7, r5 /*check condition, loop while r7 != r5*/
            beq Lsort_array_end2   /*if condition met, exit loop*/

                  ldr r8, [r4, r7, LSL #2]/*load element at address r4 +
r7x4*/
                  add r9, r7, #1         /*increment r7 value for next
element*/
                  ldr r10,[r4, r9, LSL #2]/*load element at address r4 +
r9x4*/

                  /*----------if-then------------*/
                  sort_if_eval: cmp r8, r10         /*if r8 > r10*/
                  blt sort_else                     /*branch to else if
less than*/
```

```
                        str r8, [r4, r9, LSL #2]     /*r8 stored in r10
address*/
                        str r10, [r4, r7, LSL #2]    /*r10 stored in r8
address*/

                        b end_of_sort_if
                  /*-------------else------------*/
                  sort_else:                    /*if r10 > r8*/

                        str r8, [r4, r7, LSL #2]     /*r8 stored in r8
address*/
                        str r10, [r4, r9, LSL #2]    /*r10 stored in r10
address*/

                  end_of_sort_if:
                  /*-----------------------------*/

                  add r7, r7, #1               /*increment r7*/
                  b Lsort_array_condition2     /*loop back*/

            Lsort_array_end2:
            /*-----------------------------*/

            add r6, r6, #1               /*increment r6*/
            b Lsort_array_condition1     /*loop back*/

      Lsort_array_end1:
      /*-----------------------------*/

      /*prepare call to puts*/
      ldr r0, =newline_print
      bl puts                           /*print new line*/

/*-------------------------------------------------------------------*/
      pop {r4, r5, r6, lr}   /*restore lr from stack*/
      bx lr       /*exit program*/
/*-----------------exit sort_array function---------------------*/




/*-------------------------main function-------------------------*/
.global main
main:
      push {lr} /*keep lr in the stack*/
/*-------------------------------------------------------------------*/
      /*prepare to print "unsorted array"*/
      ldr r0, =unsorted_message
      bl puts
      /*prepare call to print_array*/
      ldr r0, =my_array
      ldr r1, =len_my_array
```

```
      ldr r1, [r1]
      bl print_array

      /*prepare call to sort_array*/
      ldr r0, =my_array
      ldr r1, =len_my_array
      ldr r1, [r1]
      bl sort_array

      /*prepare to print "sorted array"*/
      ldr r0, =sorted_message
      bl puts
      /*prepare call to print_array*/
      ldr r0, =my_array
      ldr r1, =len_my_array
      ldr r1, [r1]
      bl print_array

/*-------------------------------------------------------------------*/
      mov r0, #0
      pop {lr}    /*restore lr from stack*/
      bx lr       /*exit program*/
/*-----------------------exit main function----------------------*/



/*---------external------------*/
.global puts
.global printf
/*-------------------------------------------------------------------*/
```