# Lab 3 Report

Sage Marks, Matt Krueger

3360:0001 - Embedded Systems

Professor Beichel

University of Iowa, College of Engineering

## 1. Introduction

Embedded Systems Lab 3 builds on the concepts and circuitry implemented in lab 2. Once again, we are using the 74HC595 Shift Register, 5161AS 7-Segment Display, and an enable low push button. However, the focus of this lab was working with timer/counters in the Atmel studio environment as delay loops, lookup tables, and utilizing an RPG for user inputs. The goal of the lab using the hardware described below was to create a simple electronic door lock system that accepts a 5-digit hexadecimal code. Digits selected for the code are to be scrolled through with the RPG and then selected with a push button press of less than one second. In this report, we will outline the new concepts that were employed to achieve this result and focus less on information from the prior lab.

Listed below is an exhaustive list of hardware required to replicate the lab 3 circuit. Please note that an Arduino Uno contains the microcontroller needed and is sufficient for use in lab. We opted to use the Arduino Uno for simplicity when developing the circuit.

| Hardware | Quantity | Description |
|---|---|---|
| Atmega 328P µC | 1 | Programmable µC |
| 74HC595 Shift Register | 1 | Storage of hex codes for 7-Segment display |
| 5161AS 7-Segment Display | 1 | Display current counter |
| Enable Low Push Button | 1 | Enables user input interaction with display |
| 560Ω Resistor | 8 | Resist current into 7-Segment display LEDs |
| 10KΩ Resistor | 5 | RC low pass filtering, Pull-up resistor for RPG |
| 100KΩ Resistor | 1 | Pull-up resistor for push button |
| 0.01µF Capacitor | 4 | µC Decoupling & push button RC low pass filtering |
| EVE-GA1F2012B Encoder | 1 | Rotary pulse generator |

Figure 1: Materials List

## 2. Schematic

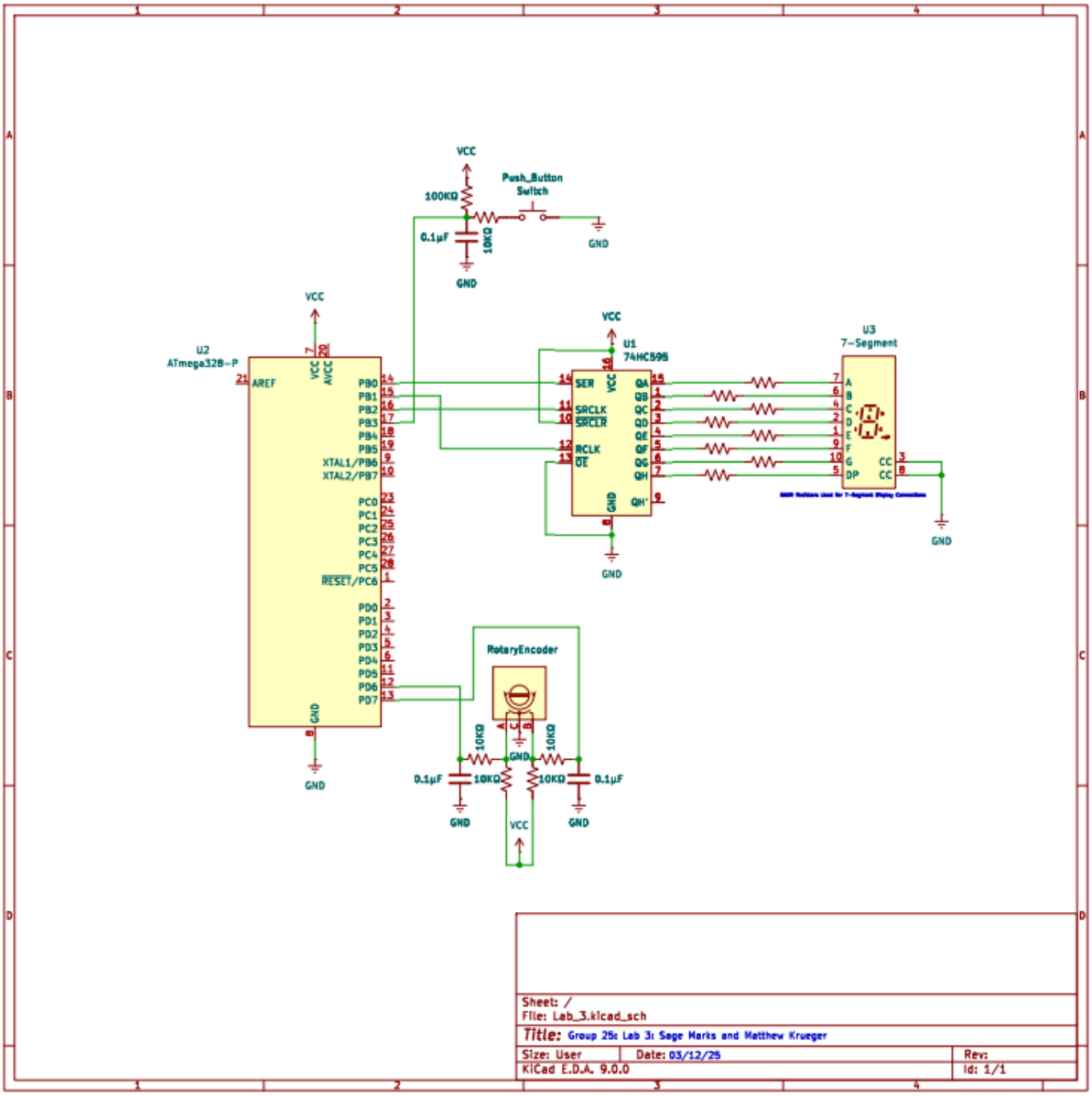


Figure 2: Electrical circuit schematic created using KiCAD

The lab 3 circuit was built off the existing 7-segment display, shift register, and push button circuit utilized in lab 2. The main difference is found in the addition of the RPG and two new input lines to the microcontroller.

The pinout of the RPG is as shown below with pin A being the A signal, pin B being the B signal and pin C being the signal ground.


The wiring of the RPG is as follows. To read the signals from pins B and A we connected pin B to PD7 and pin A to PD6 and configured these as inputs. Each pin, A and B, has a low pass RC filter connected to help with debouncing. These were created with 0.01µF capacitors and 10KΩ resistors. Then there is also a 10-kiloohm resistor that acts as a pullup resistor keeping the pins at defined states when the RPG is not being turned. Pin C is tied to the ground signal of the microcontroller.

Reading of the RPG was dependent on the signals that the RPG generates when it is turned either clockwise or counterclockwise. When turning the RPG, you can hear clicking sounds and there are defined points or grooves within the RPG called detents. These detents and the values (1 or 0) of the two signals are crucial to reading user RPG inputs. A signal being “off” is represented by 1, and a signal that is “on” is represented by a 0. When turning the RPG clockwise the B signal is the first to be on, followed by the A signal because of a 90-degree phase lag. When turning counterclockwise the A signal is the first to be on, followed by the B signal because of the 90-degree phase lag. Following a gray code system, which is reflected binary code such that only one bit at a time from 1->0 or vice versa, we can see the order that the signals will change when rotating the dial in each direction.

A clockwise rotation pattern for the signals in the order BA is as follows,

$$00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00 \ldots$$

A counterclockwise rotation pattern for the signals in the order BA is as follows,

$$00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00 \ldots$$

The difference in these signal patterns is very relevant. A detent that represents the end of a "turn" of the RPG is the signal combination 00. In the code, we compared the previous and current state of the RPG by using the "in" command. This command reads in the entire Pin D, as we are looking for inputs. Then all the other bits besides PD7 and PD6 are masked out with a bitwise and. Depending on the previous and current state we update the 7-segment display as discussed in the next section by utilizing a LUT. Before exiting the RPG logic, we save the current state as the previous state so the next comparison can be made.

## 3. Discussion

As part of the changes made from lab 2 to lab 3, the acquisition of the hex codes to represent with the 7-Segment was simplified with a lookup table (LUT). Previously, we had chained many subroutines to display each character triggered by a register tracking the current hex number 0-F. This was tedious to program and required many lines of code. A suitable fix was to incorporate a LUT named *seven_segment_codes* to store the hex codes in program memory:

```
seven_segment_codes:
.db 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07  ; 0-7
.db 0x7f, 0x6f, 0x77, 0x7c, 0x39, 0x5e, 0x79, 0x71  ; 8-f
```

Similarly, we employed another LUT to store the correct password of the electronic lock. This LUT, named *password_codes*, is also stored in program memory:

```
password_codes:
.db 0x0a, 0x0a, 0x05, 0x0c, 0x04; AA5C4
```

The assembler directive ".db" (define byte*)*, is prepended to the beginning of a sorted list of bytes. Upon loading the program, the assembler allocates and initializes these bytes at contiguous addresses in program memory, whose contents are then accessed using "LPM" (Load Program Memory) with the known memory address.

For example, to obtain the memory address for the *seven_segment_display_codes* we used a register to track the current hex displayed. Because the bytes in this LUT are sorted it is

possible to simply count the offset from the beginning of the LUT memory address. We tracked this offset using a register; upon each clockwise RPG turn, the register content was incremented and after counter-clockwise RPG turn, the register content decremented. Alone, this offset is of no use as the memory address of the wanted hex in the LUT is unknown. To find its address in memory, the Z register is utilized by loading the first two bytes of the *seven_segment_display_codes* as our program memory is word addressed and AVR architecture is 16-bit words. The Z register is a 16-bit pointer to memory locations formed by the register pair R31:R30 aliased by ZH:ZL. Note that because our LUT is defined within of the first 256 bytes, we were able to avoid using instructions to load both ZH and ZL as ZL sufficed. This is not the best practice as accessing an address in a more crowded address space (outside of the first 256 bytes) would result in an overflow into the higher byte of the Z register. This will be considered for Lab 4. Nevertheless, we can load the address of the beginning of the LUT using "LDI" (load immediate) and then add the offset to this immediate and then finally use "LPM" to get the wanted hex.

```
ldi ZL, low(seven_segment_codes << 1)

add ZL, r17

…

lpm r16, Z
```

The wanted hex from *password_codes* is obtained through a similar process using a different register. Additionally, instead of incrementing or decrementing on RPG rotations, the value in the offset register only increments at each button press less than one seconds.

The push-button functionality was also modified from lab 2 to lab 3. Now, there are only two actions distinguished by the duration of the user press.

Another important part of lab 3 was the use of timers. For all of our delay functions, we utilized a 500-microsecond base delay that is initialized once at the beginning of the program and can be called a subroutine.

To initialize this timer, we have to set a pre scalar and a starting value. Our microcontroller operates at a clock speed of 16 Mhz. That means the max delay we can reach without a pre scalar is 1/(16 x 10^6) = 16 microseconds. To reach a 500 microsecond delay the best pre scalar to go with is the Fclk/64 which corresponds to 1024 microseconds. We can get this down to 500 by selecting the correct starting position. One clock period is 4 microseconds so we can divide 500 microseconds by 4 microseconds and get 125 clock periods, then by subtracting that from 256 we get our starting position of 131 decimal.

We initialize the timer once at the start of the program by loading 83 hex (131 decimal) into the count register and then using the out command because the timer counter registers are in I/O space to timer counter register 0. Then we load the pre scalar into tmp1 by shifting in 1's to

the CS01 and CS00 positions (according to data sheet) which corresponds to the 1/64 scalar and then using out to the TCCR0B or timer counter control register.

The way that our timer works is as follows: The configurations set by the user are saved into temporary registers that can be written back each time that the subroutine is called. We check in a loop for an overflow flag that signals the timer is done, when this occurs, we are sent back to the main program with a return. When calling the subroutine again the flag is reset by writing a 1 to its location in the timer counter interrupt flag register, the specific bit we are writing is the overflow flag or the T0V0.

This base delay was utilized for checking the RPG for movement every 500 microseconds as the smallest period between rotations is 2.5 ms and creating the 4-second and 7-second delays needed for correct and incorrect code insertion. We found that using a timer for delays instead of counting clock cycles is much easier and much more reliable when it comes to getting accurate timing.

First action: user press < 1 seconds. If the user's press is less than one second long, the program checks the current digit displayed by the 7-Segment with the correct digit at the current index of the password. Recall that the current index of the password is obtained as explained previously using LUTs. If the current index is 5, then the user has guessed all digits in the code. To determine the correctness, we used register R0 to store a single bit to flag whether the user has successfully guessed all digits; if R0 is "1" the user has not entered an incorrect digit and if R0 is "0" then the user has. If R0 is 0 before password index, then it propagates throughout the entire password check process. At any password index 0-4, if the user presses the button for less than one seconds and the digit in the 7-Segment does not match the correct digit in the LUT, then bit is flagged as such. Additionally, for password indexes 0-4, following the user press for less than one second and check for match, the LUT offset stored is incremented such that the next guess is for the next index in the password. For password index 5, after the user press for less than one seconds, the last comparison takes place. After all 5 comparisons between the user's digits and the correct password digits, the status of R0 determines the next subroutine call. If the R0 bit is still "1", signifying correct password, the "correct password display" is set. The on-board LED of the Arduino Uno is turned on for 4 seconds and the decimal point "." is displayed on the 7-Segment display for this duration. During this time, the program responds to no user input. After the 4 seconds is up, the program is reset. The reset consists of restoring contents two offsets registers to the beginning of their respective LUTs, restoring R0 to "1," and setting the display on the 7-Segment display to "-" to signify the start of the program. However, If the R0 bit is "0" after all 5 password comparisons, the "incorrect password display" is set. The underscore "_" character is displayed on the 7-Segment display for 7 seconds. During this time, the program responds to no user input. After the 7 seconds is up, the program is reset in the same manner as previously described.

Second action: user press > 2 seconds. If the user's press is greater than two seconds long, then the program is reset. After releasing the button, the program simply resets. Identical to in the first action, the program reset consists of restoring contents two offsets registers to the beginning of their respective LUTs, restoring R0 to "1," and setting the display on the 7-Segment display to "-" to signify the start of the program.

## 4. Conclusion

Rome was not built in a day, nor is any million-dollar idea. Lab 3 builds on existing knowledge gained during Lab 2, extending the circuit's functionality to a rudimentary electrical locking system. We learned foundational skills of digital signal processing utilizing hardware to de-bounce and de-chatter the rotary pulse generator with pull-up resistors and low-pass filters. We also utilized software to recognize the sequencing of RPG signals using low-level AVR assembly. Although we are far off creating the next big thing, we have begun laying the mortar for the expertise required.

## 5. Appendix A: Source Code

```
; Assignment: Lab 3
; Date: 2025-03-12
; Authors: Sage Marks & Matt Krueger
;
; Description:
;      This programs a simple lock with a 7-segment display, pushbutton, and a rotary
pulse generator.
; Lookup table for 7-segment display
rjmp start
seven_segment_codes:
.db 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07     ; 0-7
.db 0x7f, 0x6f, 0x77, 0x7c, 0x39, 0x5e, 0x79, 0x71     ; 8-f

; Lookup table for password (AA5C4)
password_codes:
.db 0x0a, 0x0a, 0x05, 0x0c, 0x04, 0x00

start:
.cseg
.org 0x0000

; Port B Data Direction Configuration
sbi DDRB, 0                                                                    ; SER
sbi DDRB, 1                                                                    ; RCLK
sbi DDRB, 2                                                                    ;
SRCLK
cbi DDRB, 3                                                                    ;
pushbutton signal
sbi DDRB, 5                                                                    ; LED
on arduino board

; Port D Data Direction Configuration
cbi DDRD, 6                                                                    ; A
signal from RPG
cbi DDRD, 7                                                                ; B signal
from RPG

; Register Aliases
.def tmp1 = r23                                                                ; temporary
variables 1
.def tmp2 = r24                                                            ; temporary
variablues 2
.def count = r22                                                               ; count of
loops for timer delay
.def rpg_current_state = r21                          ; current state of RPG
.def rpg_previous_state = r20                       ; register to hold previous state of
RPG

; register aliases 16..19 were redacted for clarity as they are not as consistently
throughout the program.
; these registers are used to track the two LUTs, however during some subroutines their
function is more ambiguous.

; power_on:
;   ENTRY POINT OF THE PROGRAM
```

```
;   - initializes the timer and rpg before jumping to reset_program
;   - additionally initializes the password index to 0 and the password correctness flag
to 1 (this fixes bug where the first program loop never works)
power_on:
       rcall setup_timer
       rcall setup_rpg
       ldi r19, 0x00
       ldi tmp1, 1
       mov r0, tmp1
       rjmp start_program

; setup_rpg:
;   INITIALIZE THE ROTARY PULSE GENERATOR
;   - load PIND into rpg_previous_state (76......)
;   - mask all other bits beside wanted RPG bits 6 and 7
setup_rpg:
       in rpg_previous_state, PIND
       andi rpg_previous_state, 0xC0
       ret

; setup_timer:
;    INITIALIZE TIMER0
;      - Timer/Counter setup for 500 µs delay
;   - 8-bit timer, so we need to set the prescalar to 64
;   - atmega328p has a CPU clock speed of 16 MHz, so
;                   16,000,000/64 = 250,000
;             1/250,000 = 4 µs per cycle
;   - setting count of loops to 131, we have 125 more loops to
;     reach max 8 bit value of 255.
;             125 * 4 µs = 500 µs
;
;   TCNT0: timer counter
;   TCCR0B: timer/counter control register (CS02:00 = 011 -> fclk/64)
setup_timer:
       ldi count, 0x83
       ldi tmp1, (1<<CS01)|(1<<CS00)
       out TCNT0, count
       out TCCR0B, tmp1
       ret

; start_program:
;   INITIAL DISPLAY
;   - initialize the 7-segment display, call rpg check, and display number
;   - dislpay a dash on the 7-segment display "-" (0x40)
start_program:
       ldi r16, 0x40
       rcall display
       rcall rpg_check
       cpi r17, 0x00
       breq program_loop
       rjmp start_program

; program_loop:
;   MAIN PROGRAM LOOP
;   - update the value to be displayed on the 7-segment display via rpg interaction
;   - checks for button press & duration, and compares with password or resets program
(after 5 button clicks, code is either correct & LED shines or incorrect and program
resets after 7 seconds)
```

```
program_loop:
	rcall compute_current_seven_segment_hex
	rcall rpg_check

	; button_check:
	;   BUTTON LISTENER
	;	- if the pushbutton is pressed for <1 second, add current value stored in
r17 to password value
	;   - if the pushbutton is pressed for 1<t<2 seconds, do nothing
	;	- if the pushbutton is pressed for >2 second, reset the password value
	button_check:
		sbic PINB, 3
		rjmp program_loop

	; less_than_one:
	;	- pressed less than 1 second second, compare value with password digit
	;   - loops 0000011111010000 (2000) 2000 times * 500 microseconds = 1 second
	less_than_one:
		ldi R26, 0xd0
		ldi R27, 0x07

	 ; less_than_one_loop:
		;   - loop contained in less_than_one subroutine
		less_than_one_loop:
			rcall timer_delay_500us
			sbiw R27:R26, 1
			breq one_to_two
			sbis PINB, 3
			rjmp less_than_one_loop
			rjmp compute_hex_at_password_index

	; one_to_two:
	;	- pressed 1<t<2 seconds, do nothing
	;   - identical to less_than_one subroutine, but does not do anything if button is
pressed
	;   - loops 0000011111010000 (2000) 2000 times * 500 microseconds = 1 second (we
already know that the less than one loop has been completed)
	one_to_two:
		ldi R26, 0xd0
		ldi R27, 0x07

		; one_to_two loop:
		;   - loop contained in one_to_two subroutine
		one_to_two_loop:
			rcall timer_delay_500us
			sbiw R27:R26, 1
			breq greater_than_two
			sbis PINB, 3
			rjmp one_to_two_loop
			rjmp program_loop

	; greater_than_two:
	;	- pressed >2 seconds, reset the password value
	;   - we know that 2 seconds has already elapsed while the user is pressing the
button
	;   - simply wait until the button is released, and then reset the program
	greater_than_two:
		sbis PINB, 3
```

```
            rjmp greater_than_two
            rjmp reset_program

    ; compute_hex_at_password_index:
       ;   TRAVERSE THE PASSWORD LOOKUP TABLE
       ;   - load the value of the password at the current index
       ;   - compare with the user's selected digit
       ;   - if the values are equal, jump to correct_value_at_index
       ;   - otherwise, continue to incorrect_value_at_index
    compute_hex_at_password_index:
            ldi ZL, low(password_codes << 1)
            add ZL, r19
            lpm r18, Z
        cp r17, r18
            breq correct_value_at_index

            ; incorrect_value_at_index:
            ;   USER PASSWORD IS INCORRECT
            ;      - increment the index of the password lookup table
            ;   - set r0 to 0 (this functions as a flag to indicate that the user's
       code is incorrect)
            ;   - if the index of the password lookup table is 5, jump to
incorrect_code_display
            ;   - else, jump to program_loop to continue current attempt
            incorrect_value_at_index:
                  inc r19
                  ldi tmp1, 0
                  mov r0, tmp1
                  cpi r19, 0x05
                  breq incorrect_password_display
                  rjmp program_loop

       ; incorrect_password_display:
       ;    DISPLAY IF USER ENTERS INCORRECT CODE
       ;      - display the incorrect code pattern "_"
       incorrect_password_display:
            ldi R26, 0xb0
            ldi R27, 0x36
            ldi r16, 0x08
            rcall display

            ; incorrect_password_display_loop:
            ;      - display "_" for 7 seconds and then reset
            incorrect_password_display_loop:
                  rcall timer_delay_500us
                  sbiw R27:R26, 1
                  breq reset_program
                  rjmp incorrect_password_display_loop

       ; correct_value_at_index:
       ;   USER VALUE AT INDEX IS CORRECT
       ;      - set r0 to 1 (this functions as a flag to indicate that the user's code is
correct)
       ;   - if r0 = 0, jump to incorrect_value_at_index
       ;   - else, continue to program_loop
    correct_value_at_index:
            Mov tmp1, r0
            cpi tmp1, 0
```

```
            breq incorrect_value_at_index
        inc r19
         ldi tmp1, 1
         mov r0, tmp1
         cpi r19, 0x05
         breq check_password_correctness
            rjmp program_loop

    ; check_password_correctness:
        ;   CHECK IF THE USER PASSWORD IS CORRECT
        ;      - set r0 to 1 (this functions as a flag to indicate that the user's code is
correct)
        ;   - if r0 = 1, jump to LED_ON, else continue to reset_program
        ;   - triggered after 5 button presses regardless of correctness
    check_password_correctness:
        mov tmp1, r0
         cpi tmp1, 0x01
         breq correct_password

    ; reset_program:
        ;   RESET THE PROGRAM
        ;      - set r0 to 1 (true initially... user hasnt entered the wrong code yet)
        ;   - reset the index of the password lookup table to 0
        ;   - jump to power_on to restart the program
        ;
        ;   'ldi r17, 0x01'
        ;   - no significance of 0x01 other than it isnt 0x00... needed to fix error when
        ;       restarting resetting at 0. Because r17 would be set to 0x00, it would
branch
        ;     after the cpi in start_program, skipping the display of the "-".
        ;       This fixes the issue; the user can now stay at "-" until rotary is moved
cw.
    reset_program:
            ldi tmp1, 1
            mov r0, tmp1
            ldi r19, 0x00
            ldi r17, 0x01
            rjmp power_on

    ; correct_password:
        ;
        ;      - display the LED on the arduino board
        ;   - display a "." on the 7-segment display
        ;   - only displayed if the user's code is correct
        ;   - runs for 4 seconds
    correct_password:
        ldi R26, 0x40
            ldi R27, 0x1f
        sbi PORTB, 5
            ldi r16, 0x80
            rcall display

            ; correct_password_loop:
            ;      - display the LED on the arduino board for 4 seconds
            correct_password_loop:
                rcall timer_delay_500us
                sbiw R27:R26, 1
                breq  LED_off
```

```
                rjmp correct_password_loop

        ; LED_off:
        ;       - turn off the LED on the arduino board
        LED_off:
                cbi PORTB, 5
                rjmp reset_program

; rpg_check:
   ;       - check the state of the RPG
   ;    - if the state has not changed, jump to no_change
   ;    - otherwise, jump to check_state
   ;    - rotary pulse generator has 3 pins: A, B, and C
   ;    - A and B are used to determine the direction of the rotation
   ;    - A and B are set to pin 6 and 7 of the arduino board
   ;    - C is used to reset the program
rpg_check:
        rcall timer_delay_500us
        in rpg_current_state, PIND
        andi rpg_current_state, 0xC0
        cp rpg_current_state, rpg_previous_state
        breq no_change
        cpi rpg_current_state, 0x00
        breq check_state
        rjmp save_rpg_state

   ; check_state:
   ;       - check the state of the RPG
   ;    - if the state is 0x80, jump to cw_check  (10000000 - A is high)
   ;    - if the state is 0x40, jump to ccw_check (01000000 - B is high
   ;    - otherwise, jump to save_rpg_state
   check_state:
        cpi rpg_previous_state, 0x80
        breq cw_check
        cpi rpg_previous_state, 0x40
        breq ccw_check
        rjmp save_rpg_state

   ; ccw_check:
   ;       - check the state of the RPG
   ;    - if the state is 0x00, jump to counter_clockwise
   ;    - otherwise, jump to ret
   ccw_check:
        cpi rpg_current_state, 0x00
        breq counter_clockwise
        ret

   ; cw_check:
   ;       - check the state of the RPG
   ;    - if the state is 0x00, jump to clockwise
   ;    - otherwise, jump to ret
   cw_check:
        cpi rpg_current_state, 0x00
        breq clockwise
        ret

   ; counter_clockwise:
   ;       - check the state of the RPG
```

```
        ;   - if the state is 0x40, jump to save_rpg_state
        ;   - otherwise, jump to ret
        ;   - note - if the current value displayed is '0', and the user continues to
rotate clockwise, remain at 0
        counter_clockwise:
                cpi r16, 0x40
                breq save_rpg_state
                dec r17
                cpi r17, 0xff
                breq display_min_bound
                rjmp save_rpg_state

        ; clockwise:
        ;       - check the state of the RPG
        ;   - if the state is 0x40, jump to firstMovement
        ;   - otherwise, jump to ret
        ;   - note - if the current value displayed is 'f', and the user continues to
rotate clockwise, remain at 'f'
        clockwise:
                cpi r16, 0x40
                breq first_movement
                inc r17
                cpi r17, 0x10
                breq display_max_bound
                rjmp save_rpg_state

        ; first_movement:
        ;       - set the value of r17 to 0
        ;   - jump to save_rpg_state
        first_movement:
                ldi r17, 0x00
                rjmp save_rpg_state

        ; display_min_bound:
        ;       - set the value of r17 to 0
        ;   - jump to save_rpg_state
        display_min_bound:
                ldi r17, 0x00
                rjmp save_rpg_state

        ; display_max_bound:
        ;       - set the value of r17 to 0xf
        ;   - jump to save_rpg_state
        display_max_bound:
                ldi r17, 0x0f
                rjmp save_rpg_state

        ; save_rpg_state:
        ;       - save the current state of the RPG
        ;   - move the current state of the RPG to the previous state
        ;   - this is used to determine the direction of the rotation
        save_rpg_state:
                mov rpg_previous_state, rpg_current_state

        ; no_change:
        ;   RPG STATE HAS NOT CHANGED
        ;       - if this code block is reached, simply return
        no_change:
```

```
            ret

; compute_current_seven_segment_hex:
;   TRAVERSE THE SEVEN SEGMENT DISPLAY LOOKUP TABLE
;      - find the value to be displayed on the 7-segment display
;   - lookup the value in the seven_segment_codes table
;   - load the value into r16
compute_current_seven_segment_hex:
       ldi ZL, low(seven_segment_codes << 1)
       add ZL, r17

; display_call:
;   DISPLAY THE VALUE ON THE 7-SEGMENT DISPLAY (driver)
;      - calls the function display after loading z into r16 (current hex code) before
returning to loop
display_call:
       lpm r16, Z
       rcall display
       ret

; display:
;   DISPLAY THE VALUE ON THE 7-SEGMENT DISPLAY
;      - display the current value on the 7-segment display
;   - utilizes the stack to save the value of r16 and r17 and the status register
display:
       push r16
       push r17
       in r17, SREG
       push r17
       ldi r17, 8

  ; rotate_bits:
  ;   SERIAL INPUT OF DATA TO 74HC595 SHIFT REGISTER
  ;   - subroutine for 74hc595 shift register to shift in the value of the hex code to be
displayed on the 7-segment display
  ;   - shift the value of r16 to the left
  ;   - if the carry flag is set, set the value of PORTB.0
  ;   - otherwise, clear the value of PORTB.0
  ;   - this runs 8 times to rotate all 8 bits of the hex code into the shift register
  rotate_bits:
    rol r16
    brcs set_ser_in_1
    cbi PORTB, 0
       rjmp shift_register_out

  ; set_ser_in_1:
  ;   PULSE 1 TO SERIAL INPUT OF 74HC595 SHIFT REGISTER
  ;   - subroutine for 74hc595 shift register to shift in the value of the hex code to be
displayed on the 7-segment display
  ;   - set the value of PORTB 0
  set_ser_in_1:
    sbi PORTB, 0

  ; shift_register_out:
  ;   OUTPUT DATA TO 74HC595 SHIFT REGISTER
  ;   - subroutine for 74hc595 shift register to shift in the value of the hex code to be
displayed on the 7-segment display
```

```
  ;   - pulse srclk and rclk to shift in the value of the hex code to be displayed on the
7-segment display.
  ;   - first shifts all digits in from r16 using rotate_bit loop, then stores them (upon
storage, because OE active, there is output from shift register... electrical circuit
handles the rest).
  ;   - restore the values in registers pushed onto stack
  shift_register_out:
    sbi PORTB, 2
    cbi PORTB, 2
    dec r17
    brne rotate_bits
    sbi PORTB, 1
    cbi PORTB, 1
       pop r17
       out SREG, r17
       pop r17
       pop r16
       ret

;timer_delay_500us:
;   DELAY FOR 500 MICROSECONDS
;   - delay for 500 microseconds using timer0 of the atmega328p
;
;      relevant registers:
;             - TCCR0: timer counter control register - setting modes of timer/counter
;       - TIFR0: timer counter interrupt flag register - if TOV0 set (1), then the timer
is overflown
;             - TCNT0: timer counter register - counts up with each pulse to the value
loaded into it
timer_delay_500us:
       in tmp1, TCCR0B
       ldi tmp2, 0x00
       out TCCR0B, tmp2
       in tmp2, TIFR0
       sbr tmp2, 1<<TOV0
       out TIFR0, tmp2
       out TCNT0, count
       out TCCR0B, tmp1

       ; wait_for_overflow:
       ;   - wait for the value of TOV0 to be set (overflown)
       wait_for_overflow:
              in tmp2, TIFR0
              sbrs tmp2, TOV0
              rjmp wait_for_overflow
              ret
```

## 6. Appendix B: References

Beichel, Reinhard. *Embedded Systems Lab 2, ECE:3360. The University of Iowa, 2025* <https://uiowa.instructure.com/courses/248357/files/29567265?module_item_id=8134634>

Beichel, Reinard. *Embedded Systems, Rotary Pulse Generators and Lab 3. The University of Iowa, 2025* <*https://uiowa.instructure.com/courses/248357/files/29778930?module_item_id=8162158* >

Beichel, Reinard. *Embedded Systems, Lecture 8: Timers. The University of Iowa, 2025* <*https://uiowa.instructure.com/courses/248357/files/29853139?module_item_id=8165716* >

Components101. *7 Segment Display. 22 September 2019.* <https://components101.com/displays/7-segment-display-pinout-working-datasheet>

Pighixxx. *The Definitive Arduino Uno Pinout Diagram. May 5, 2013.* <https://uiowa.instructure.com/courses/248357/files/29320694?module_item_id=8042318>

Texas Instruments. *SNx4HC595 8-Bit Shift Registers With 3-State Output Register. September 2015*. <https://www.ti.com/lit/ds/symlink/sn74hc595.pdf>

XLITX. *5161AS Datasheet* <http://www.xlitx.com/datasheet/5161AS.pdf>