University of California, Riverside

EE 128 Fall 2024

Sensing and Actuation For Embedded Systems

Mini Project Report

8 December 2024

Justin Fababier

Matthew Henson

Contents

Project Description

The "Digital Door Lock" project aims to simulate the functionality of a digital lock using the FRDM-K64F single-board microcontroller. It utilizes general-purpose input/output (GPIO) pins, pulse-width modulation (PWM), 2-phase bipolar stepper motor control, and interrupt service routines.

Initially, the system is in a locked state, with the password stored in a predefined 4-element array. User input is captured through a 4-by-4 membrane keypad, with each key press stored in a separate 4-element array. To provide feedback, a passive buzzer emits a 1 kHz tone (using PWM) for 250 ms with each button press, while a blue LED lights up.

Once all four user inputs are entered, the system compares the user-input array with the stored password array. If they match, the system transitions to the unlocked state, activating the stepper motor to rotate 32-steps clockwise, simulating unlocking. A green LED is also illuminated. If the input does not match, the user-input array is cleared, and the stepper motor remains stationary. In the unlocked state, the system can be relocked by pressing the '#' key, causing the stepper motor to rotate 32-steps counterclockwise.

The system relies on interrupts as a key component of its multitasking framework. A Periodic Interrupt Timer (PIT) generates interrupts at intervals specified by `GCD_PERIOD`. Each interrupt triggers the `PIT0_IRQHandler` Interrupt Service Routine (ISR), acting as the system scheduler. The ISR tracks the elapsed time for all tasks and executes the state machine functions of tasks whose elapsed time meets or exceeds their designated period. This interrupt-driven approach ensures precise timing, task synchronization, and efficient operation by eliminating the need for continuous polling.

System Design

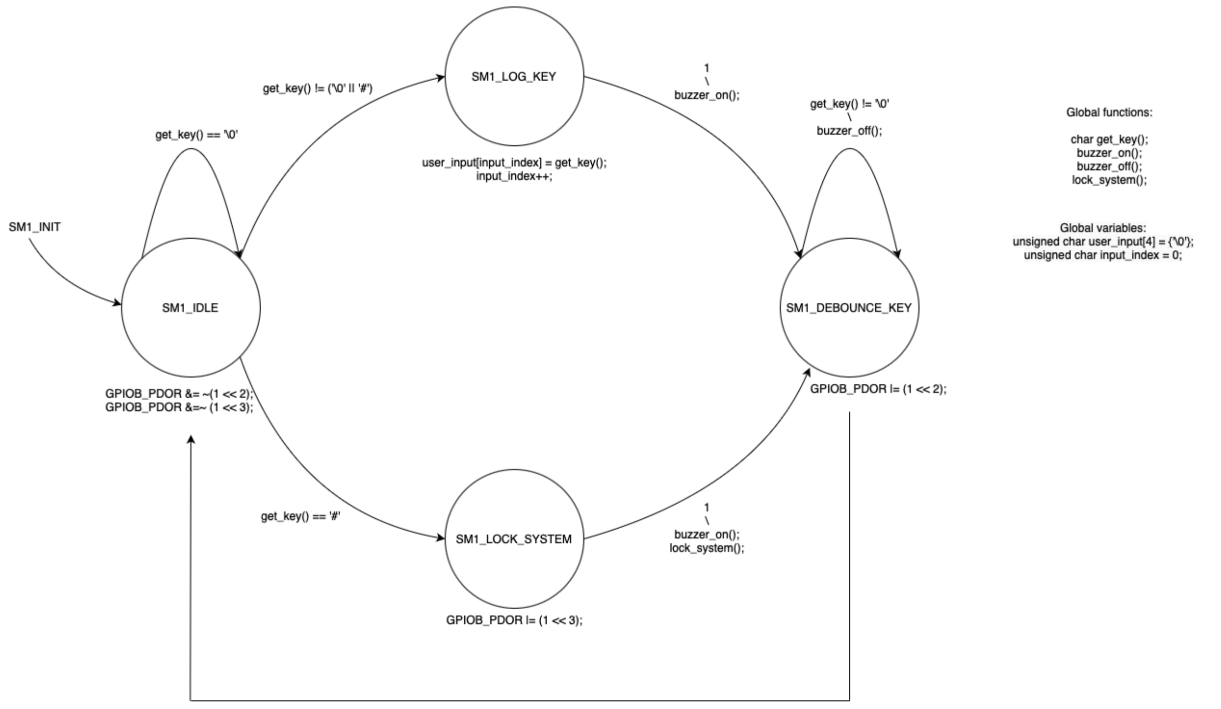
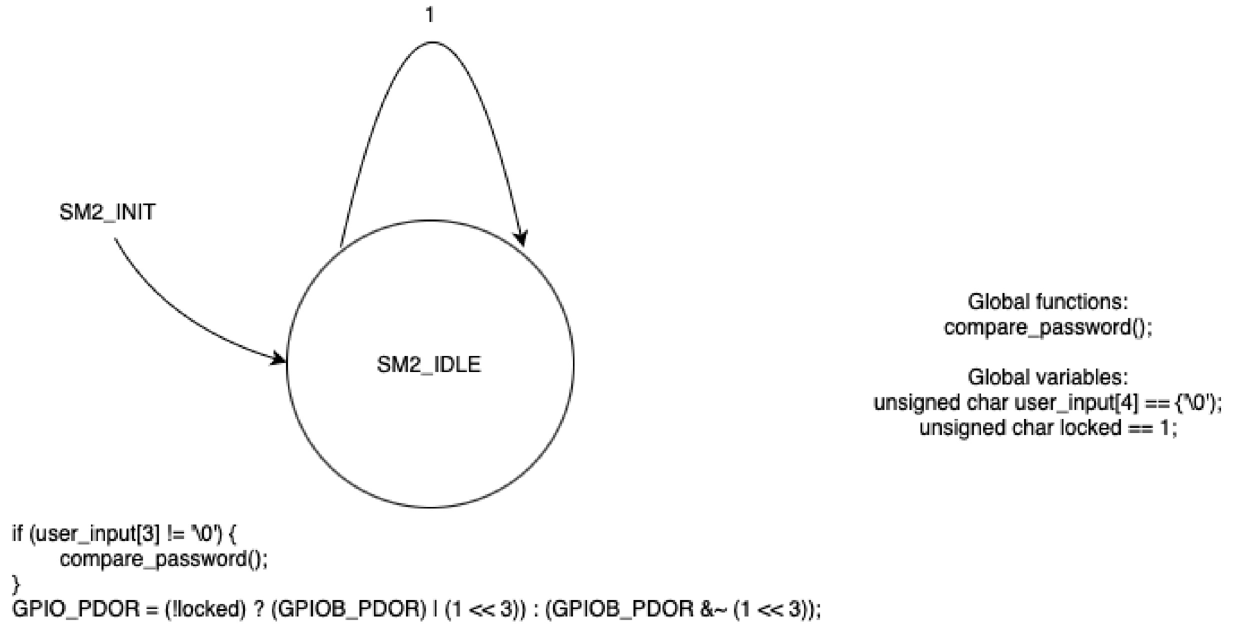


Figure 1: State Machine 1 diagram.
Period: 250 ms
(Note: global variable 'locked' is accessed via 'lock_system()')

State Machine 1 (SM1_Tick in the code) handles logging user input from the 4-by-4 membrane keypad. User input is obtained through the `get_key()` function, which returns a null character (`\0`) if no input is detected. The system begins in a locked state, and button debouncing is handled in a dedicated state, such as SM1_DEBOUNCE_KEY. When the system is unlocked, the user can press the '#' key on the keypad to relock it.

State Machine 1 also controls the project's red and blue LEDs. The blue LED illuminates when any key other than '#' is pressed, whereas the red LED illuminates when the '#' key is pressed. Additionally, State Machine 1 interfaces the passive buzzer via Mealy actions (See: Discussions).

System Design



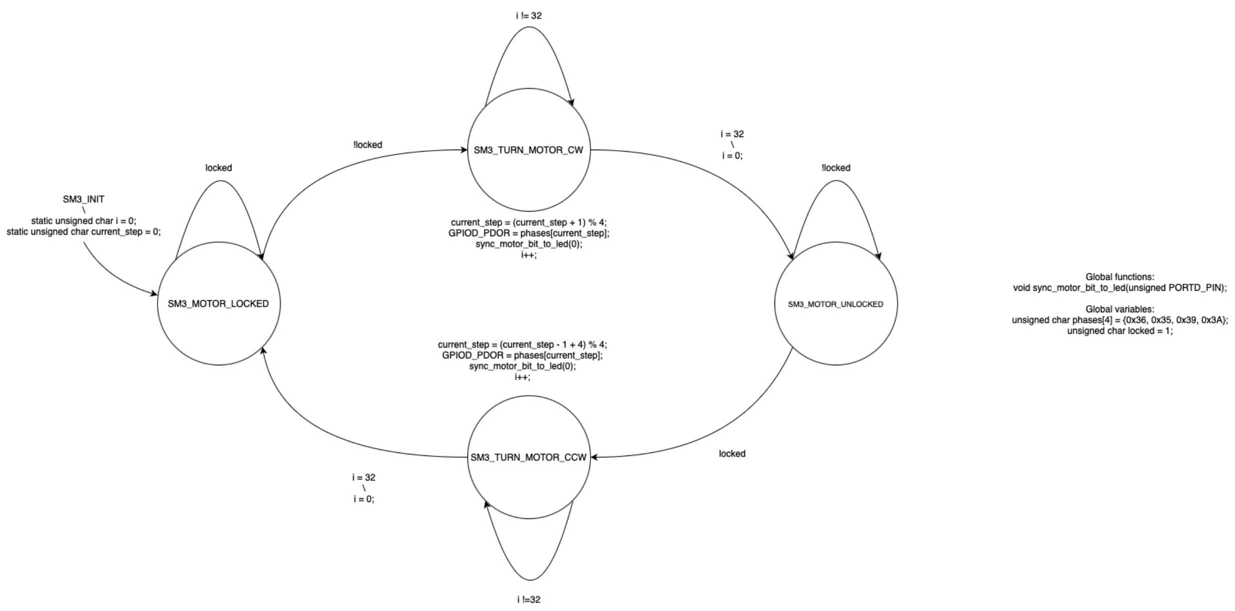
```
if (user_input[3] != '\0') {
      compare_password();
}
GPIO_PDOR = (!locked) ? (GPIOB_PDOR) | (1 << 3)) : (GPIOB_PDOR &~ (1 << 3));
```

Figure 2: State Machine 2 diagram.
Period: 1000 ms

State Machine 2 (SM2_Tick in the code) implicitly manages password comparison and system locking. The `compare_password()` function performs the comparison, while its nested functions `lock_system()` and 'unlock_system()' handle the locking implementation. For more information on 'compare_password()', see the Implementation Details section.

Additionally, State Machine 2 handles LED interfacing for the lock's state, which affects a green LED based on the variable 'locked'. If the 'locked' is true, the green LED is powered on. Conversely, if 'locked' is false, the green LED is powered off.

System Design



```
static unsigned char i = 0;
static unsigned char current_step = 0;
```

```
current_step = (current_step + 1) % 4;
GPIOD_PDOR = phases[current_step];
sync_motor_bit_to_led(0);
i++;
```

```
current_step = (current_step - 1 + 4) % 4;
GPIOD_PDOR = phases[current_step];
sync_motor_bit_to_led(0);
i++;
```

```
Global functions:
void sync_motor_bit_to_led(unsigned PORTD_PIN);

Global variables:
unsigned char phases[4] = {0x36, 0x35, 0x39, 0x3A};
unsigned char locked = 1;
```

Figure 3: State Machine 3 diagram.
Period: 15 ms

State Machine 3 (SM3_Tick in the code) controls the operation of the stepper motor. Initially, the system assumes a locked state, so the motor starts in the SM3_MOTOR_LOCKED state by default. When the global variable `locked` changes from `true` to `false`, the state transitions to SM3_TURN_MOTOR_CW, which causes the motor to rotate 32-steps clockwise. After this, the motor enters the SM3_MOTOR_UNLOCKED state and remains idle. Conversely, when `locked` changes from `false` to `true`, the state transitions to SM3_TURN_MOTOR_CCW, making the motor rotate 32-steps counterclockwise before returning to the SM3_MOTOR_LOCKED state.

During both SM3_TURN_MOTOR_CW and SM3_TURN_MOTOR_CCW, the function `sync_motor_bit_to_led(0)` is called. This function reads the value of a Port D pin and reflects the value onto a white LED. In this implementation, the function reads the value of Port D's pin 0. When the motor is rotating, the white LED flashes 32 times within a 480 ms period.

Pinout Details

4-by-4 Membrane Keypad:

- Rows (0-3): Connected to PC0 to PC3, configured as GPIO output pins.
- Columns (0-3):
    - Connected to PC4, PC5, PC7, and PC8, configured as GPIO input pins.
    - Pull-up resistors enabled for PC4, PC5, PC7, PC8.
    - Note: See Page 284 of *K64 Sub-Family Reference Manual* for pull-up resistor setup.

LEDs:

- Red LED: Connected to PB2.
- Green LED: Connected to PB3.
- Blue LED: Connected to PB10.
- White LED: Connected to PB11.

Stepper Motor (Small Motor):

- V_ss: Connected to a 5V power supply
- V_s: Connected to a 9V power supply
- IN[1:4]: Connected to PD0 to PD3.
- IN1 (Orange wire): Connected to PD0.
- IN2 (Green wire): Connected to PD1.
- IN3 (Yellow wire): Connected to PD2.
- IN4 (Red wire): Connected to PD3.
- EN[1:2]: Connected to PD4 and PD5.
    - EN1 (Grey wire): Connected to PD4.
    - EN2 (Brown wire): Connected to PD5.
- OUT[1:4]:
    - A1 (Yellow wire): Connected to OUT1.
    - A2 (White wire): Connected to OUT2.
    - B1 (Red wire): Connected to OUT3.
    - B2 (Blue wire): Connected to OUT4.

Passive Buzzer:

- Connects to: PC10.
- Desired signal: 1 kHz PWM.
    - FTM3_MOD = 10499.
    - FTM3_C6V = 5250.
    - Prescaler: Set to 2.

Implementation Details

The task structure forms the backbone of the cooperative multitasking system, encapsulating each task's finite-state machine state, period, elapsed time, and function pointer. Tasks are organized within a task array, allowing centralized management and streamlined iteration. The macro `NUM_TASKS` defines the total number of tasks in the system, enabling easy scalability by adjusting the task count without significant code modifications. Task execution periods are specified using constants such as `SM1_PERIOD` and `SM2_PERIOD`, with `GCD_PERIOD` serving as the base period for scheduling.

The PIT0 ISR is essential for time-based task management, iterating through tasks to update elapsed times and invoking tick functions when their periods have elapsed. The `TimerSet(unsigned long period)` function configures the Periodic Interval Timer (PIT) module to generate interrupts at a specified interval, while `TimerOn()` activates the timer and enables NVIC interrupts. State enumerations and declarations define possible states and associated tick functions for each task, creating a framework for implementing task behaviors. Tick functions handle state transitions and task actions. The `main()` function initializes tasks, configures the timer, and enters an infinite loop where tasks are executed in response to PIT interrupts. This design provides a scalable and modular approach to multitasking for embedded systems. For further details, refer to *K64 Sub-Family Reference Manual* (Chapter 41, Pages 1083–1094).

Justin Fababier's synchronous FSM template for the FRDM-K64F is available on GitHub. The code is derived from the synchronous finite-state machine template provided by the EE 120B Spring 2024 instructional team for the Atmega328p hardware platform.
https://github.com/justinfababier/FRDM-K64F-SynchSM-Template/blob/main/frdmk64f_synchsm_template.c

Implementation Details

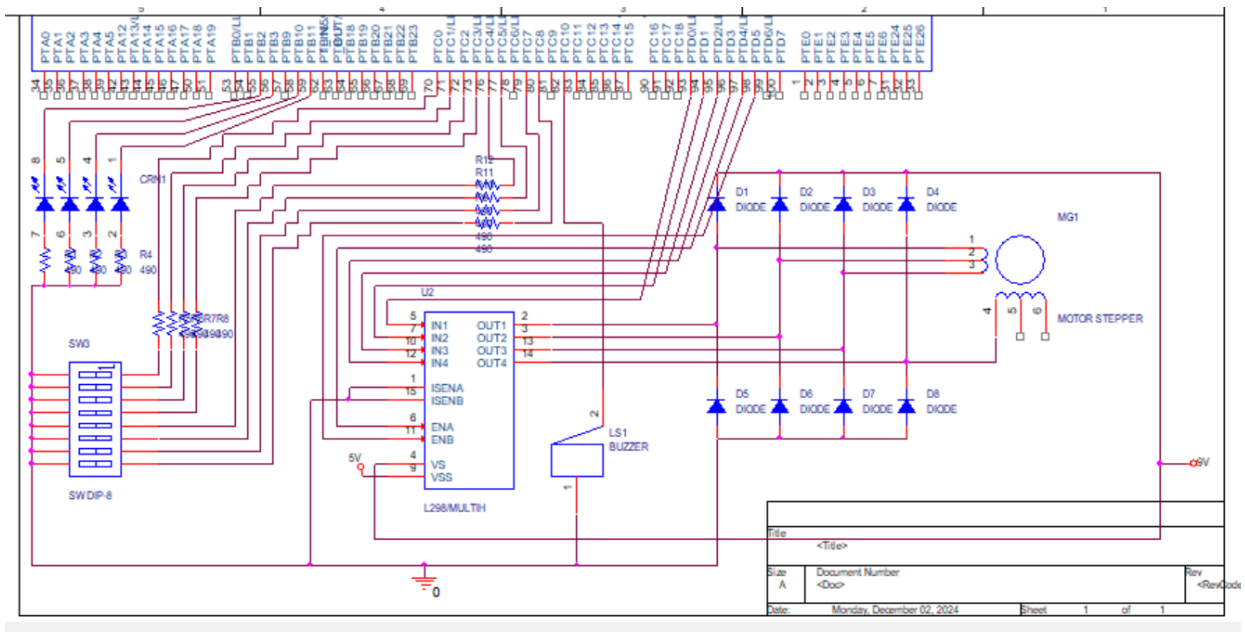


Figure 4: PSpice circuit diagram of the digital door lock project.

Implementation Details

Below are the functions for locking and unlocking the system.

```
// Lock system
void lock_system(void) {
        locked = 1;                 // Set system to locked state
        input_index = 0;            // Reinitialize input_index to 0
        memset(user_input, '\0', sizeof(user_input));  // Reset user_input to all '\0'
}

// Unlock system
void unlock_system(void) {
        locked = 0;                         // Set system to unlocked state
        input_index = 0;                    // Reinitialize input index to 0
        GPIOB_PDOR &= ~(1 << 2);    // Turn off red LED
        GPIOB_PDOR |= (1 << 3);                 // Turn on green LED
}
```

‘compare_password()’ is called within SM2_IDLE. If the for-loop completes itself, ‘unlock_system()’ is called. If the inner if-statement returns true, ‘lock_system()’ is called and the function exits.

```
// Compare user-input to password
void compare_password(void) {
        unsigned char i;

        // Check if user_input matches the password
        for (i = 0; i < 4; i++) {
                if (user_input[i] != password[i]) {
                        // If any character doesn't match, keep the system locked and indicate
error
                        lock_system(); // Reset lock
                        return;        // Exit the function immediately
                }
        }

        // If all characters match, unlock the system
        unlock_system();
}
```

Implementation Details

The `get_key()` function detects a pressed key on a 4-by-4 membrane keypad by sequentially activating one row at a time (setting it to `LOW`) while leaving the other rows inactive (`HIGH`). For each active row, it checks the input state of all columns, which are connected to GPIO pins configured as inputs. If a column reads `LOW`, it indicates a key in the active row and corresponding column is pressed. The function then uses a `keypad[row][col]` mapping array to identify and return the specific key value. If no key is pressed after checking all rows and columns, the function returns a null character ('\0').

```
// Keypad input capture function
char get_key(void) {
        // Define the column pin mapping: indices correspond to physical pins PC4, PC5, PC7,
PC8
        unsigned char column_pins[] = {4, 5, 7, 8};

        // Loop through each row
        for (unsigned char row = 0; row < 4; row++) {
                // Set the current row to LOW (active), others HIGH (inactive)
                GPIOC_PDOR = ~(1 << row) & 0x0F;

                // Check each column
                for (unsigned char col = 0; col < 4; col++) {
                        if ((GPIOC_PDIR & (1 << column_pins[col])) == 0) {  // Column is LOW
(key pressed)

                                // Return the detected key from the keypad map
                                return keypad[row][col];
                        }
                }
        }

        // No key pressed, return null character
        return '\0';
}
```

Testing/Evaluation

Testing and development took place in WCH 128 and at team members' personal residences. The equipment utilized in WCH 128 includes oscilloscopes and multimeters for signal analysis, and voltage supplies to power the L298N motor driver module. During the development process, LEDs were regularly used to facilitate interfacing the 4-by-4 membrane keypad, the 2-phase bipolar stepper motor, and various software functions.

Discussions

This project demonstrates the features of an embedded system by simulating the operation of a digital door lock. While the design effectively showcases fundamental concepts, it does not reflect the power efficiency typical of real-world digital locks. Key contributors to this inefficiency include continuous GPIO polling, the FRDM-K64F microcontroller running at its standard clock speed, and the L298N driver module's 12V power requirement for the stepper motor.

One notable improvement for the project would be enhancing the readability of State Machine 1 in the code. Currently, it heavily relies on Mealy actions, as shown in the state machine diagram (see: System Design). Readability can be improved by moving each state's Mealy actions into its corresponding state action block within the code.

An enhancement to the project involves better control of the stepper motor's operation. In the SM3_MOTOR_LOCKED and SM3_MOTOR_UNLOCKED states, the motor should be powered off by setting the ENA and ENB pins to a digital low, effectively disabling them. Incidentally, not doing so killed our 9V battery when the system was idle. Additionally, it is recommended to set the pins connected to IN[1:4] to low, helping to minimize unnecessary current output by the FRDM-K64F board. This can be accomplished by masking Port D's output register using the following operation: **GPIOD_PDOR &= ~0x3F;**.
In the original code, the stepper motor remains powered on in these states, despite not rotating. By implementing this change, we prevent the L298N driver module from supplying power to the OUT[1:4] pins, ensuring that the motor only draws current when needed. It also ensures that bits 6 and beyond remain unaffected, preserving the functionality of other pins on Port D.

Another significant improvement would be modifying the manipulation method of Port D's output register. The current implementation affects all bits of Port D's output register, despite State Machine 3 only needing to control the stepper motor's pins. This can be improved by explicitly controlling pins 0-5 of Port D, ensuring that if other components are connected to pins 6 and higher; the current setup will not cause unexpected issues. This can be accomplished by masking Port D's output register with the following operation:
GPIOD_PDOR = (GPIOD_PDOR & ~0x3F) | (phases[current_step] & 0x3F);.

Roles & Responsibilities

Justin and Matthew wrote and tested the code for the project throughout the four-week project. During the first few weeks, Justin wrote a synchronous finite-state machine template used for this project. Matthew created the PSpice circuit diagram. Both contributed to assembling the physical circuit, with development taking place during and outside of lab hours. Both members were present the day of the lab project presentations.

Technical Problems Encountered

During the project, various challenges were faced, such as interfacing the 4-by-4 membrane keypad, generating the appropriate PWM signal for the passive buzzer, configuring the stepper motor wiring, and deciding whether to implement a synchronous FSM. While creating the PSpice circuit diagram, we decided to use a 8-pin DIP Switch to represent the membrane keypad. This decision was made because a proper membrane keypad part could not be found in the component library.

A major issue involved operating the stepper motor. It remained non-functional up until the presentation day due to a damaged L298N driver module missing its heatsink being in our lab kit. During the day of the lab presentation, it was replaced with a functional L298N driver module, allowing the stepper motor to work as intended.

Conclusion

The project was successful, with the code effectively implementing GPIO, PWM, stepper motor operation, and interrupt service routines. All components, including the 4-by-4 membrane keypad, passive buzzer, and 2-phase bipolar stepper motor, functioned as intended during the day of the lab presentations. The project simulates digital door lock behavior, as intended.

Images

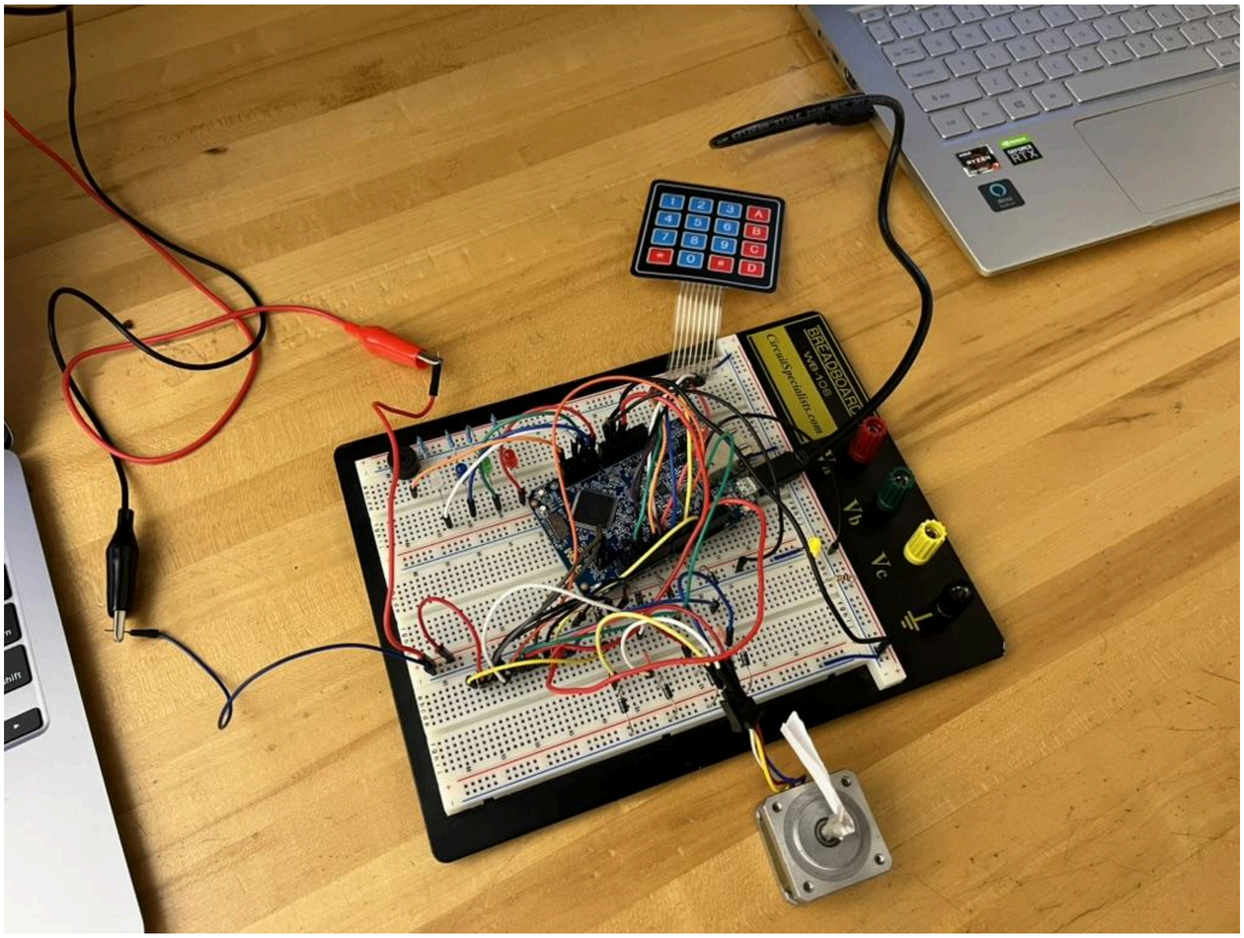


Figure 5: Breadboard circuit of digital door lock, post-presentation.
(Note: L298N driver module is not connected to circuit in this image.)

Code

This code is available for viewing on Github.
https://github.com/justinfababier/EE-128-Mini-Project

Below is the completed code for this project.

```
/*******************************************************************************
* Project Name: EE 128 Mini Project - Door Lock
* File Name: main.c
* Author: J. Fababier, M. Henson
* Submission Date: [December 06, 2024]
* Description:
*   This project simulates a simple digital door lock's behavior using the
following components:
*   - 1x 4-by-4 membrane keypad for user input,
*   - 1x passive buzzer for audio feedback,
*   - 1x 2-phase bipolar stepper motor to simulate a locking mechanism.
*
* Notes:
*   [Any additional notes, assumptions, or limitations regarding the program.]
*
*******************************************************************************/

#include "fsl_device_registers.h"

// Macro definition for the number of tasks in the system
#define NUM_TASKS 3

// Task structure definition
typedef struct _task {
   signed char state;               // Task's current state
   unsigned long period;            // Task period (in ms)
   unsigned long elapsedTime;       // Time elapsed since last task tick (in ms)
   int (*TickFct)(int);             // Pointer to the task's state machine
function
} task;

// Array to hold the tasks in the system
task tasks[NUM_TASKS];
```

```
// Task periods (in ms). Adjust based on desired behavior.
const unsigned long GCD_PERIOD = 5; // Common divisor period
const unsigned long SM1_PERIOD = 250;    // State machine 1 period
const unsigned long SM2_PERIOD = 1000;   // State machine 2 period
const unsigned long SM3_PERIOD = 15;     // State machine 3 period

// PIT0 Interrupt Service Routine (ISR)
void PIT0_IRQHandler(void) {
   // Clear the interrupt flag for PIT channel 0
   PIT_TFLG0 |= PIT_TFLG_TIF_MASK;

   // Iterate over all tasks and handle execution
   for (unsigned char i = 0; i < NUM_TASKS; i++) {
       if (tasks[i].elapsedTime >= tasks[i].period) {
           // Execute the task's tick function and update its state
           tasks[i].state = tasks[i].TickFct(tasks[i].state);
           // Reset elapsed time for the task
           tasks[i].elapsedTime = 0;
       }
       // Increment elapsed time by the GCD_PERIOD
       tasks[i].elapsedTime += GCD_PERIOD;
   }
}

// Function to configure the PIT timer for the desired period
void TimerSet(unsigned long period) {
   // Enable clock for PIT module
   SIM_SCGC6 |= SIM_SCGC6_PIT_MASK;

   // Enable the PIT module
   PIT_MCR &= ~PIT_MCR_MDIS_MASK;

   // Calculate the timer load value for a 1 ms interrupt
   uint32_t timerLoadValue = (period * (SystemCoreClock / 1000)) - 1;
   PIT_LDVAL0 = PIT_LDVAL_TSV(timerLoadValue);

   // Enable interrupts for PIT channel 0
   PIT_TCTRL0 |= PIT_TCTRL_TIE_MASK;
}

// Function to start the PIT timer
```

```
void TimerOn() {
   // Start the PIT timer on channel 0
   PIT_TCTRL0 |= PIT_TCTRL_TEN_MASK;
   // Enable the PIT interrupt in the NVIC
   NVIC_EnableIRQ(PIT0_IRQn);
}

// Global variables
unsigned char password[] = {'1', '1', '1', '1'};    // Password as characters
unsigned char user_input[4] = {'\0'};           // Initialize all elements to 0
unsigned char input_index = 0;               // Index for the user input
unsigned char locked = 1;                // Locked state: 0 = unlocked, 1 =
locked
unsigned char phases[4] = {0x36, 0x35, 0x39, 0x3A}; // Stepper motor phases

// Keypad mapping
static char keypad[4][4] = {
       {'1', '2', '3', 'A'},
       {'4', '5', '6', 'B'},
       {'7', '8', '9', 'C'},
       {'*', '0', '#', 'D'}
};

// Keypad input capture function
char get_key(void) {
   // Define the column pin mapping: indices correspond to physical pins PC4,
PC5, PC7, PC8
   unsigned char column_pins[] = {4, 5, 7, 8};

   // Loop through each row
   for (unsigned char row = 0; row < 4; row++) {
       // Set the current row to LOW (active), others HIGH (inactive)
       GPIOC_PDOR = ~(1 << row) & 0x0F;

       // Check each column
       for (unsigned char col = 0; col < 4; col++) {
           if ((GPIOC_PDIR & (1 << column_pins[col])) == 0) {  // Column is LOW
(key pressed)

               // Return the detected key from the keypad map
               return keypad[row][col];
```

```
            }
        }
    }

    // No key pressed, return null character
    return '\0';
}

// Lock system
void lock_system(void) {
    locked = 1;      // Set system to locked state
    input_index = 0;    // Reinitialize input_index to 0
    memset(user_input, '\0', sizeof(user_input));  // Reset user_input to all
'\0'
}

// Unlock system
void unlock_system(void) {
    locked = 0;          // Set system to unlocked state
    input_index = 0;         // Reinitialize input index to 0
    GPIOB_PDOR &= ~(1 << 2);    // Turn off red LED
    GPIOB_PDOR |= (1 << 3);     // Turn on green LED
}

// Compare user-input to password
void compare_password(void) {
    unsigned char i;

    // Check if user_input matches the password
    for (i = 0; i < 4; i++) {
        if (user_input[i] != password[i]) {
            // If any character doesn't match, keep the system locked and
indicate error
            lock_system();  // Reset lock
            return;     // Exit the function immediately
        }
    }

    // If all characters match, unlock the system
    unlock_system();
}
```

```c
// Passive buzzer off, PWM mode disabled
void buzzer_off(void) {
   FTM3_MODE = 0x5;         // Enable FTM3
   FTM3_MOD = 10499;        // period
   FTM3_C6SC = 0x00;        // PWM active low
   FTM3_C6V = 5250;         // pulse width
   FTM3_SC = (1 << 3) | 1;      // system clock; prescale 2
}

// Passive buzzer on, PWM mode enabled
void buzzer_on(void) {
   FTM3_MODE = 0x5;         // Enable FTM3
   FTM3_MOD = 10499;        // period
   FTM3_C6SC = 0x28;        // PWM active high
   FTM3_C6V = 5250;         // pulse width
   FTM3_SC = (1 << 3) | 1;      // system clock; prescale 2
}

// Read the value of the specified GPIOD pin
void sync_motor_bit_to_led(unsigned int PORTD_PIN) {
   // Read the value of the specified GPIOD pin
   unsigned int bit_value = (GPIOD_PDOR & (1 << PORTD_PIN)) >> PORTD_PIN;

   // Control the LED on PB11 based on the bit value
   GPIOB_PDOR = bit_value ? (GPIOB_PDOR | (1 << 11)) : (GPIOB_PDOR & ~(1 <<
11));
}

// State machine enumerations and declarations
enum SM1_Tick { SM1_INIT, SM1_IDLE, SM1_LOG_KEY, SM1_LOCK_SYSTEM,
SM1_DEBOUNCE_KEY };                // States for state machine 1
enum SM2_Tick { SM2_INIT, SM2_IDLE, SM2_LOCK_SYSTEM, SM2_DEBOUNCE_KEY };
// States for state machine 2
enum SM3_Tick { SM3_INIT, SM3_MOTOR_LOCKED, SM3_MOTOR_UNLOCKED,
SM3_TURN_MOTOR_CW, SM3_TURN_MOTOR_CCW };     // States for state machine 3
int SM1_Tick(int state);     // Tick function for state machine 1
int SM2_Tick(int state);     // Tick function for state machine 2
int SM3_Tick(int state);     // Tick function for state machine 3

// State machine 1 tick function - Manage password logging
```

```
int SM1_Tick(int state) {
   // State transitions
   switch (state) {
   case SM1_INIT:
       state = SM1_IDLE;
       break;

   case SM1_IDLE:
       if (get_key() == '\0') {
           state = SM1_IDLE;
       }
       else if (get_key() == '#') {
           state = SM1_LOCK_SYSTEM;
       }
       else {
           state = SM1_LOG_KEY;
       }
       break;

   case SM1_LOG_KEY:
       buzzer_on();
       state = SM1_DEBOUNCE_KEY;
       break;

   case SM1_LOCK_SYSTEM:
       buzzer_on();
       lock_system();
       state = SM1_DEBOUNCE_KEY;
       break;

   case SM1_DEBOUNCE_KEY:
       buzzer_off();
       if (get_key() != '\0') {
           state = SM1_DEBOUNCE_KEY;
           break;
       }
       state = SM1_IDLE;
       break;

   default:
       break;
```

```
    }

    // State actions
    switch (state) {
    case SM1_INIT:
        break;

    case SM1_IDLE:
        GPIOB_PDOR &= ~(1 << 2);     // Turn off red LED
        GPIOB_PDOR &= ~(1 << 10);    // Turn off blue LED
        break;

    case SM1_LOG_KEY:
        user_input[input_index] = get_key();
        input_index++;
        break;

    case SM1_LOCK_SYSTEM:
        GPIOB_PDOR |= (1 << 2);
        break;

    case SM1_DEBOUNCE_KEY:
        GPIOB_PDOR |= (1 << 10);     // Turn on blue LED
        break;

    default:
        break;
    }

    return state;
}

// State machine 2 tick function - Handle password comparison, LED interfacing
int SM2_Tick(int state) {
    // State transitions
    switch (state) {
    case SM2_INIT:
        state = SM2_IDLE;
        break;

    case SM2_IDLE:
        state = SM2_IDLE;
        break;

    default:
        break;
    }

    // State actions
    switch (state) {
    case SM2_INIT:
        break;

    case SM2_IDLE:
        if (user_input[3] != '\0') {
            compare_password();
        }
        GPIOB_PDOR = (!locked) ? (GPIOB_PDOR | (1 << 3)) : (GPIOB_PDOR & ~(1 <<
3)); // Handle green LED
        break;

    default:
        break;
    }

    return state;
}

// State machine 3 tick function - Stepper motor operation
int SM3_Tick(int state) {
    static unsigned char i = 0;      // Variable to track steps
    static unsigned char current_step = 0;  // Current step index

    // State transitions
    switch (state) {
    case SM3_INIT:
        state = SM3_MOTOR_LOCKED;
        break;

    case SM3_MOTOR_LOCKED:
        if (locked) {
            state = SM3_MOTOR_LOCKED;
            break;
        }
        if (!locked) {
            state = SM3_TURN_MOTOR_CW;
        }
        break;

    case SM3_MOTOR_UNLOCKED:
        if (!locked) {
            state = SM3_MOTOR_UNLOCKED;
        }
        if (locked) {
            state = SM3_TURN_MOTOR_CCW;
        }
        break;

    case SM3_TURN_MOTOR_CW:
        if (i == 32) {
            i = 0;
            state = SM3_MOTOR_UNLOCKED;
        }
        else {
            state = SM3_TURN_MOTOR_CW;
        }
        break;

    case SM3_TURN_MOTOR_CCW:
        if (i == 32) {
            i = 0;
            state = SM3_MOTOR_LOCKED;
        }
        else {
            state = SM3_TURN_MOTOR_CCW;
        }
        break;

    default:
        break;
    }

    // State actions
```

```
   switch (state) {
   case SM3_INIT:
       break;

   case SM3_MOTOR_LOCKED:
       break;

   case SM3_MOTOR_UNLOCKED:
       break;

   case SM3_TURN_MOTOR_CW:
       current_step = (current_step + 1) % 4;  // Increment step in circular
manner
       GPIOD_PDOR = phases[current_step];
       sync_motor_bit_to_led(0);       // Toggle white LED
       i++;
       break;

   case SM3_TURN_MOTOR_CCW:
       current_step = (current_step - 1 + 4) % 4;  // Decrement step in circular
manner
       GPIOD_PDOR = phases[current_step];
       sync_motor_bit_to_led(0);           // Toggle white LED
       i++;
       break;

   default:
       break;
   }

   return state;
}

// Main function
int main(void) {
   // Refer to "EE 128 Mini Project Pinout" for pinout details
   // Enable clock for PORTB, PORTC, PORTD
   SIM_SCGC5 |= (SIM_SCGC5_PORTB_MASK | SIM_SCGC5_PORTC_MASK |
SIM_SCGC5_PORTD_MASK);  // Enable clock for PORTB, PORTC, PORTD

   // Configure PORTB GPIO pins
```

```
PORTB_GPCLR = 0x0C0C0100;   // Configure PB2, PB3, PB10, PB11 as GPIO
GPIOB_PDDR |= 0x0C0C;       // Set PB2, PB3, PB10, PB11 as outputs
GPIOB_PDOR |= 0x0000;       // Set PB2-PB3 as LOW

// Configure PORTC GPIO pins
PORTC_GPCLR = 0x01BF0100;   // Configure PC0-PC3, PC4, PC5, PC7, PC8 as GPIO
GPIOC_PDDR |= 0x000F;       // Set PC0-PC3 as outputs (rows)
GPIOC_PDDR &= ~0x01B0;      // Set PC4, PC5, PC7, PC8 as inputs (columns)
GPIOC_PDOR |= 0x0000;       // Set outputs to LOW
PORTC_PCR4 |= PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;  // Enable pull-up on PC4
PORTC_PCR5 |= PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;  // Enable pull-up on PC5
PORTC_PCR7 |= PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;  // Enable pull-up on PC7
PORTC_PCR8 |= PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;  // Enable pull-up on PC8

// Configure PC10 for PWM output (FTM3_CH6)
// PC10 generates 1 kHz PWM signal
// Refer to: - buzzer_off(), buzzer_on()
SIM_SCGC3 |= SIM_SCGC3_FTM3_MASK;   // Enable clock for FTM3
PORTC_PCR10 = 0x300;            // Port C Pin 10 as FTM3_CH6 (ALT3)

// Configure PORTD GPIO pins
PORTD_GPCLR = 0x003F0100;   // Configure pins PD0-PD5
GPIOD_PDDR |= 0x003F;       // Set pins PD0-PD5 as outputs
GPIOD_PDOR &= ~0x003F;      // Set outputs to low

// Configure Task 1
unsigned char i = 0;
tasks[i].state = SM1_INIT;      // Set initial state
tasks[i].period = SM1_PERIOD;   // Set task period
tasks[i].elapsedTime = tasks[i].period; // Initialize elapsed time
tasks[i].TickFct = &SM1_Tick;   // Set tick function

// Configure Task 2
i++;
tasks[i].state = SM2_INIT;      // Set initial state
tasks[i].period = SM2_PERIOD;   // Set task period
tasks[i].elapsedTime = tasks[i].period; // Initialize elapsed time
tasks[i].TickFct = &SM2_Tick;   // Set tick function

// Configure Task 3
i++;
```

```
    tasks[i].state = SM3_INIT;       // Set initial state
    tasks[i].period = SM3_PERIOD;    // Set task period
    tasks[i].elapsedTime = tasks[i].period; // Initialize elapsed time
    tasks[i].TickFct = &SM3_Tick;    // Set tick function

    // Configure and start the timer
    TimerSet(GCD_PERIOD);            // Set timer period
    TimerOn();                       // Start timer

    // Run program indefinitely
    while (1) {
        // Infinite loop
    }

    // Program should never reach here
    return 0;
}
```