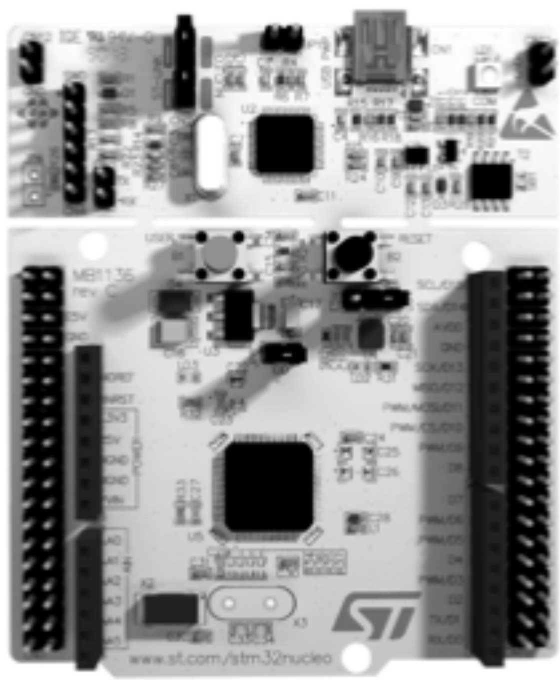
PROJECT #1 – DIGITAL LOCKBOX

Cal Poly SLO | EE 329 - 01 | Professor Paul Hummel



Ethan Clark Najmy

Spring Quarter | April 20, 2022

[1] Photo taken from STM32 Lab Manual

1. Behavior Description

The digital lockbox created in this project consists of an LCD display, a 4x4 keypad, and an STM32L4A6ZGT6 board. The project functions as a digital lock whose status is displayed on the LCD display and on-board LED on the STM32 board. The system can be unlocked or reprogramed using the keypad digits 0-9.

The user will enter in the default 4-digit combination and press # and the device will unlock indicated on the LCD and by an on-board LED on the STM32 board. The user can then lock the system again by pressing * or reprogram the system with their own 4-digit combination by pressing #.

At any time while entering a pin, the user may press * to clear the digits on the display. If a wrong pin is entered, the user will see a wrong pin message on the LCD display and will need to press # to return to the main screen.

When reprograming, the system will not allow a pin greater than or less than 4 digits to be entered. When a pin of 4 digits has been entered, the user may press # to save this new pin.

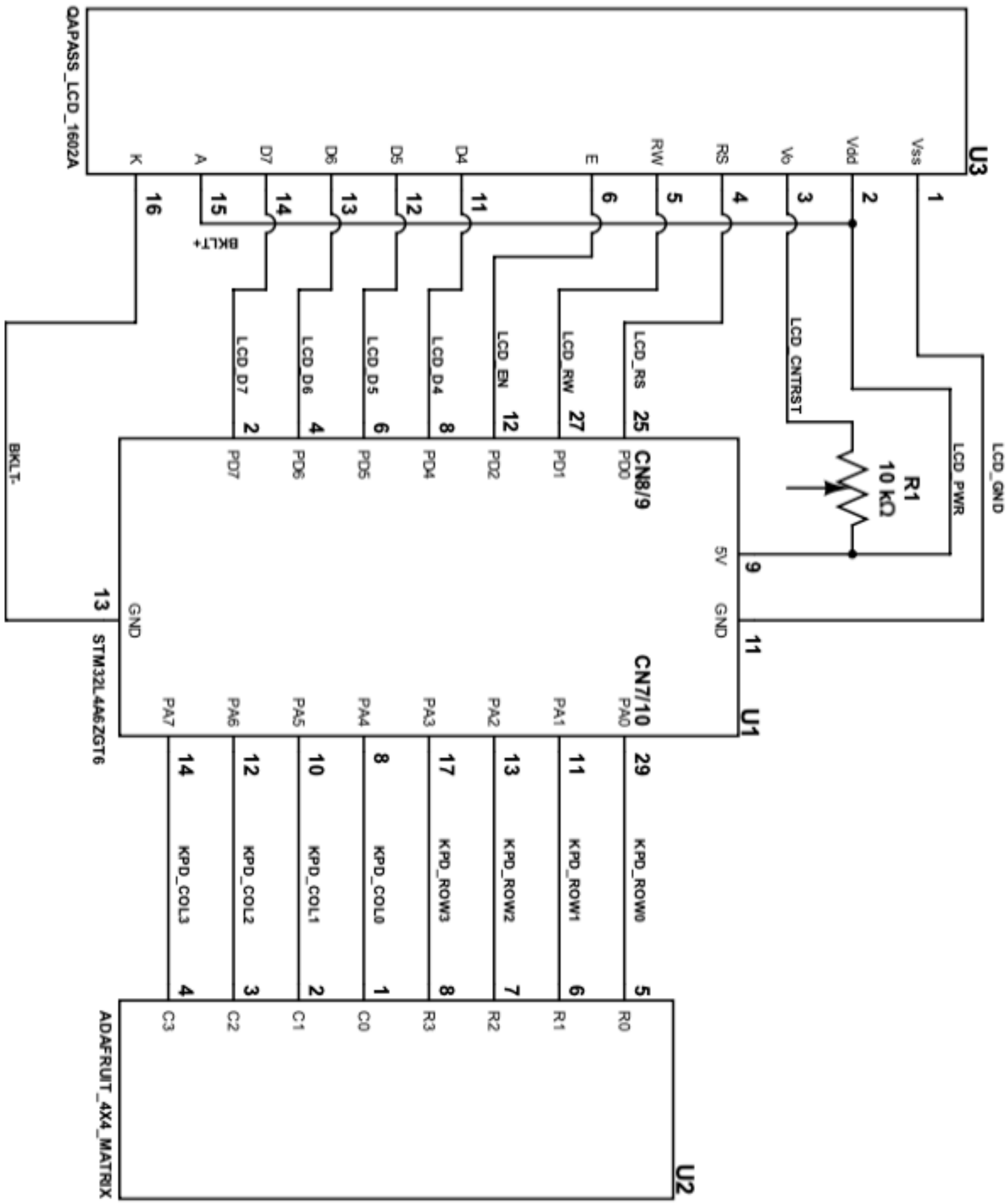
2. System Specifications

Table 1. System Specifications

Power Specifications	
Supply Voltage	-0.3V – 4V
Current Draw	150mA (max.)
Power Consumption	0.6W (max.)
Power Connection	Micro USB cable (not included)
Keypad Specifications	
Total Keys	16
Usable Keys	12
Default Combination	1234
LCD Specifications	
Required Pin Length	4 digits
LCD Screen Size	16x2

3. System Schematic

Figure 1. System Schematic



4. Software Architecture

4.1. Overview

The system functions through the use of a software-implemented Finite State Machine (FSM). When powered on, the system initializes the FSM, keypad, LCD, onboard LED, and enters the FSM. The system will stay in the FSM while powered on and will only switch between the 5 FSM states.

4.2. Initialization

The system initializes the FSM through the use of the `typedef` keyword (source code can be found in *Appendix B*), creating a custom data type and creating the individual states. Next, key variables are defined for storing button presses, keeping count, the user's pin, and the default pin. The default pin is initially defined with a value of 1234. The keypad, LCD, and onboard LED are initialized through individual functions: `keypad_init()`, `LCD_init()`, and `LED_init()`. In each of these functions, the GPIO pins that each perepherial is connected to are initialized and configured as appropriate.

Figure 2. LCD Initialization Flowchart

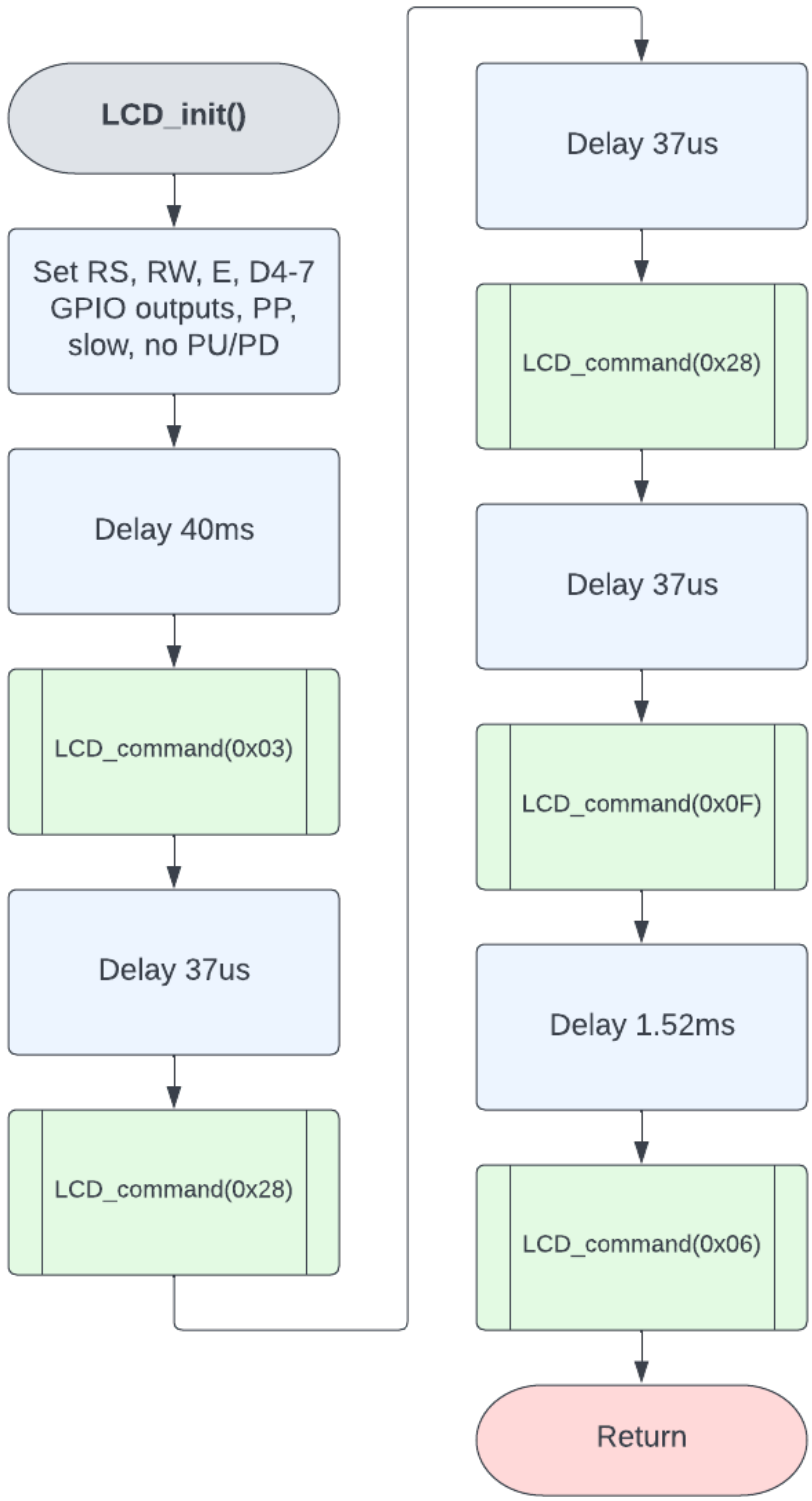
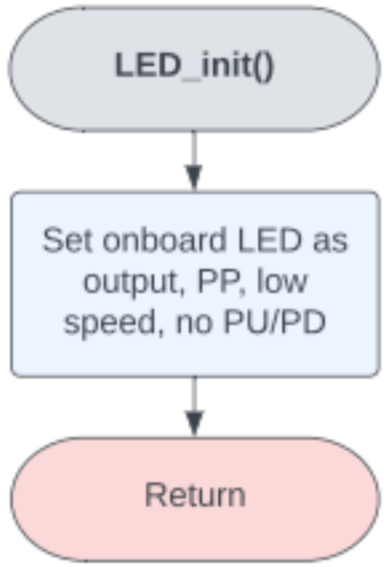


Figure 3. LED Initialization Flowchart

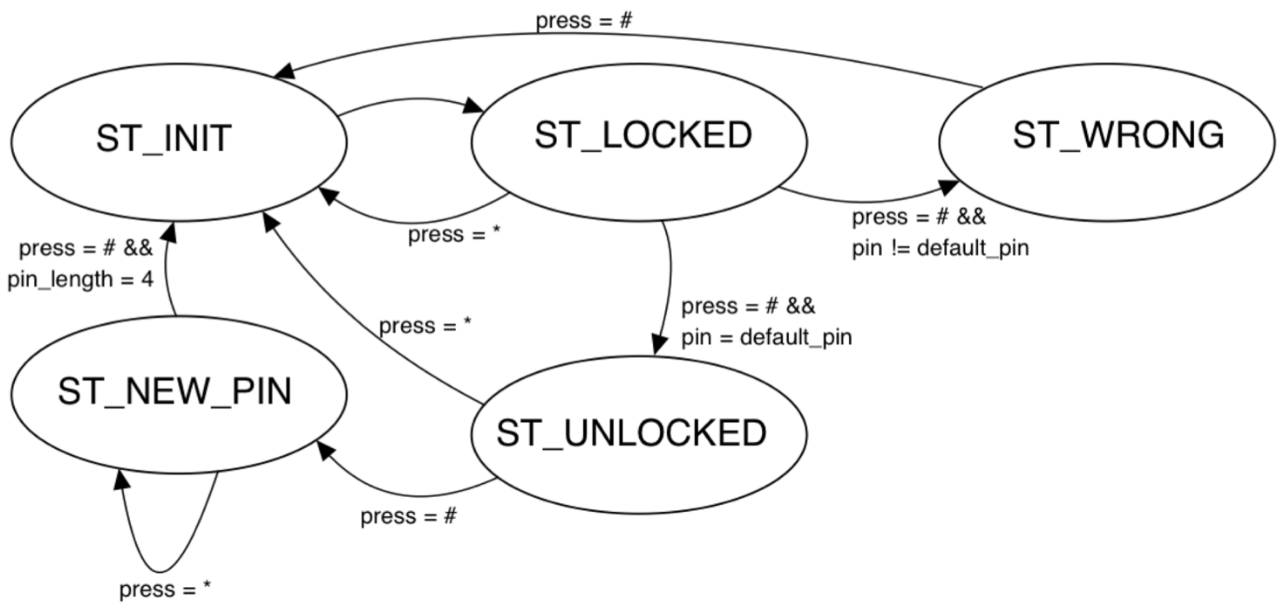


** Keypad initialization flowchart omitted due to derivation by Professor Hummel*

4.3. FSM States

The FSM in this system has 5 states: ST_INIT, ST_LOCKED, ST_UNLOCKED, ST_WRONG, ST_NEW_PIN. Each state represents an important part of the lockbox.

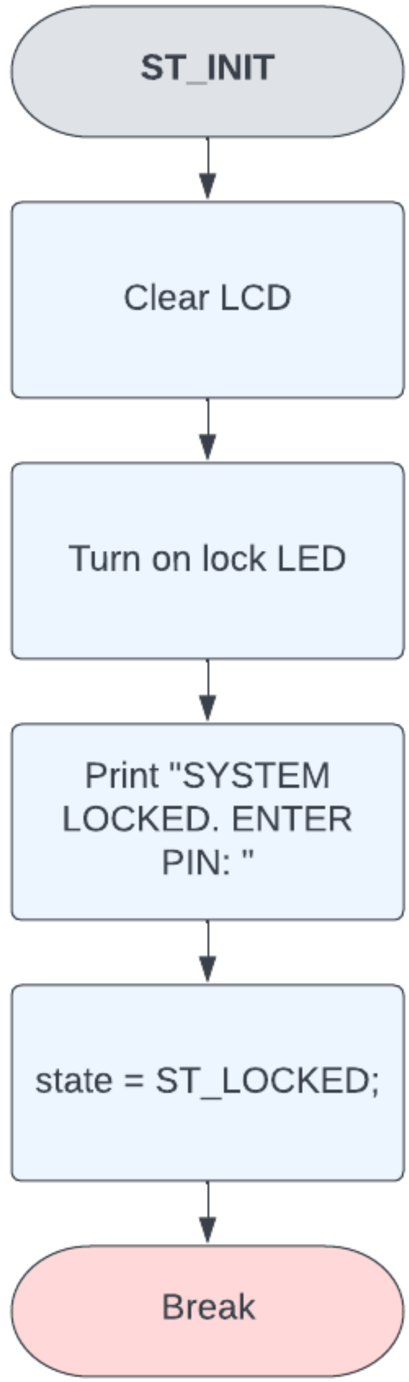
Figure 4. FSM State Diagram



4.3.1. ST_INIT

The system begins in ST_INIT where it turns on the LED, prints to the LCD: "`SYSTEM LOCKED. ENTER PIN: `", and resets important variables. In this section and throughout the code we have two variables: `count` and `pin`. The variable `count` is used to keep track of how many numbers have been entered, allowing for the conversion of individual button presses into a 4-digit long integer that is the pin number. The variable `pin` is used for storing the pin the users have entered and comparison to the default pin.

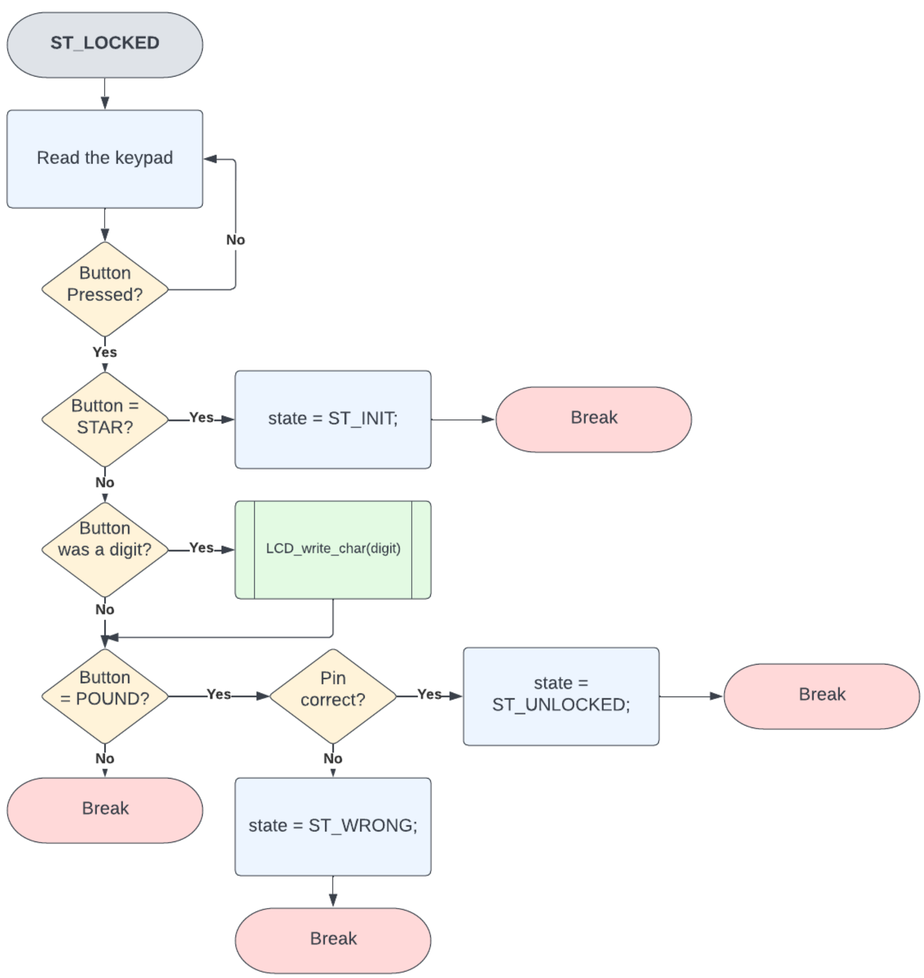
Figure 5. ST_INIT Flowchart



4.3.2. ST_LOCKED

After ST_INIT, the system automatically moves into ST_LOCKED. In this state it waits for a button to be pressed. If a button was pressed it checks to see if it was * or # or a digit. If it was * the system will reset, clear, move back into ST_INIT, then automatically go back into ST_LOCKED. If it was a digit, it will print the digit on the screen while updating the variable `pin`. Using `count` and a x10 multiplier, the digits entered will be turned into a 4-digit long integer. If # was pressed, the system will check to see if the pin entered is equal to the default pin. If so, it will transition into ST_UNLOCKED. If not, it will transition into ST_WRONG.

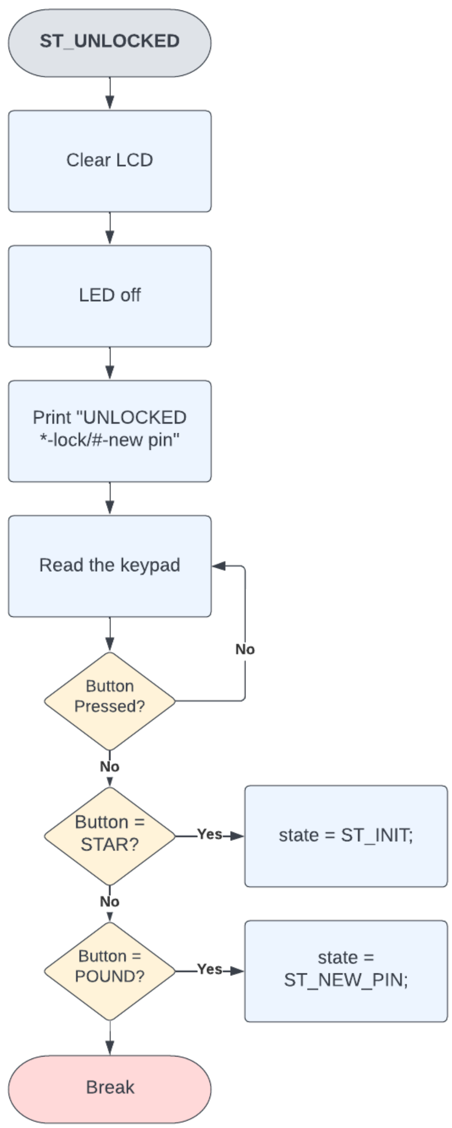
Figure 6. ST_LOCKED Flowchart



4.3.3. ST_UNLOCKED

When in ST_UNLOCKED, the LED will turn off and the system will display "`UNLOCKED *-lock/#-new pin`" prompting the user to either lock the system by pressing * or entering a new pin by pressing #. The system will wait for a button press and move to either ST_LOCKED or ST_NEW_PIN depending on which button was pressed.

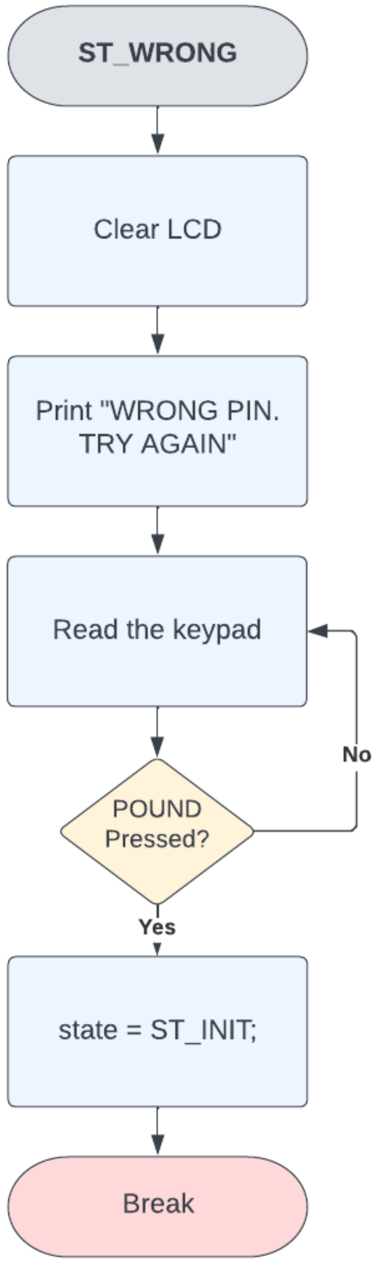
Figure 7. ST_UNLOCKED Flowchart



4.3.4. ST_WRONG

In ST_WRONG, the system will keep the LED on, indicating it is still locked, while displaying "`WRONG PIN. TRY AGAIN.`" It will then wait for the user to press # to return to ST_INIT and reset.

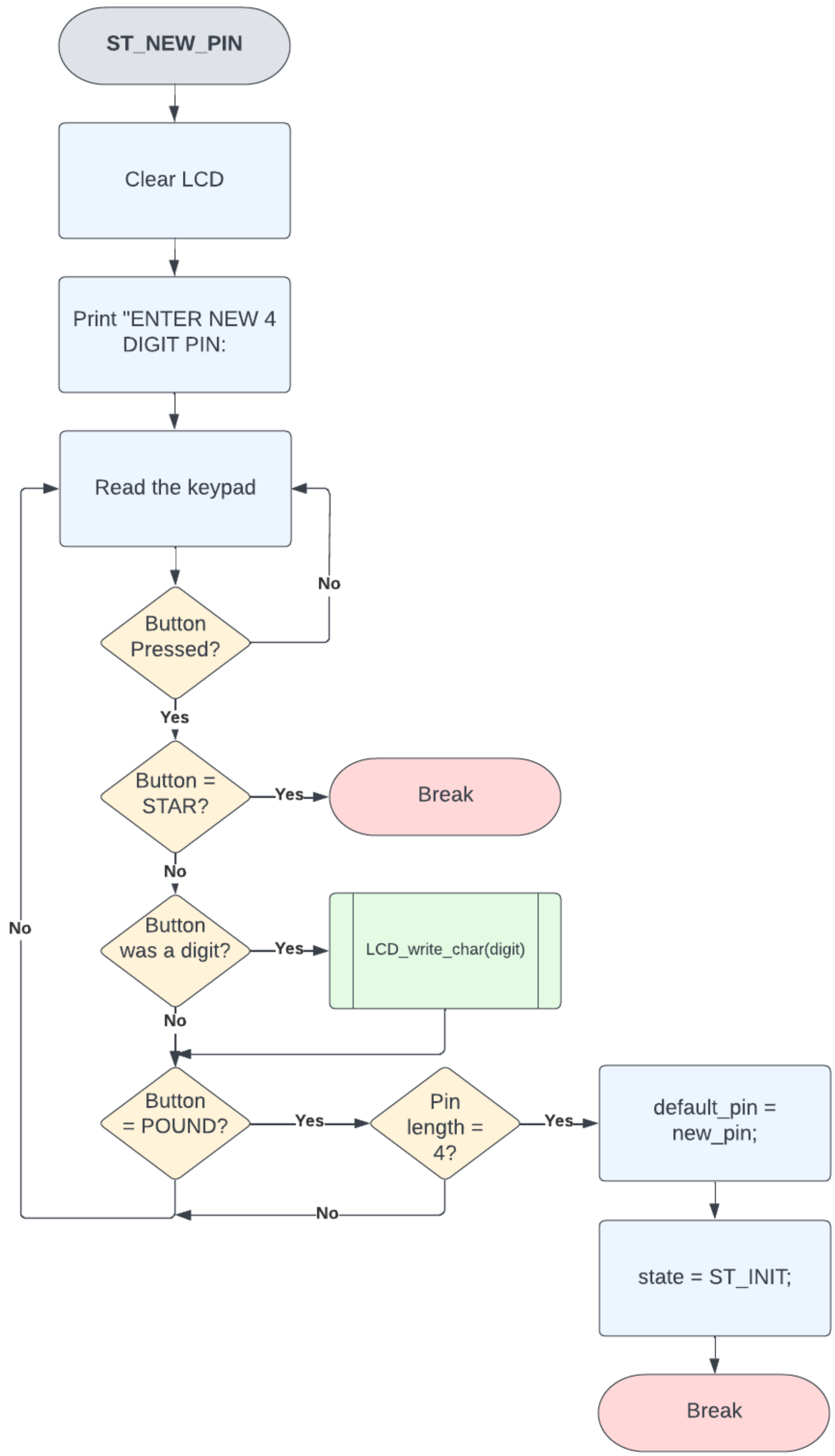
Figure 8. ST_WRONG Flowchart



4.3.5. ST_NEW_PIN

In ST_NEW_PIN, the system resets `pin` and `count` to zero in order to keep accurate track of the digits entered. It will then print "`ENTER NEW 4 DIGIT PIN: `" while displaying each key press on the display. Again, the user may press * to clear the display of the digits they typed. The system will wait until the # key is pressed and check to see if exactly 4 digits have been entered. If 4 digits have been entered, the system will save the pin and return to ST_INIT. If more or less than 4 digits have been entered, the system will not change screens or states and the user will have to enter 4 digits.

Figure 9. ST_NEW_PIN Flowchart



4.4. Subroutines and Functions

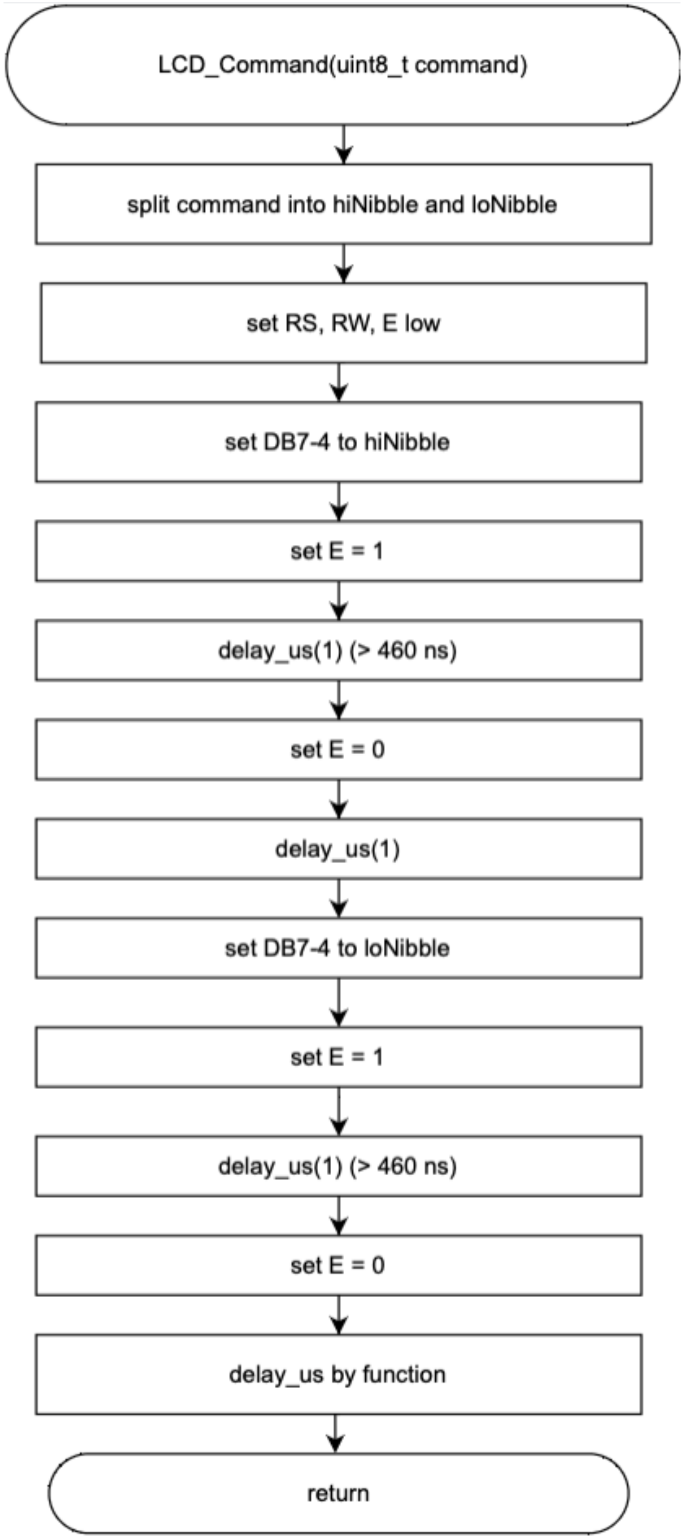
In each state, subroutines/functions are used to operate different aspects of the system. For example, printing to the LCD requires the use of `LCD_command()`, `LCD_write_char()`, and `convertNum()`. In this section each function will be explained.

4.4.1. LCD Functions

`void LCD_command(uint8_t command)`

This function operates by sending commands to the LCD microcontroller in 2 parts: `hiNibble` and `loNibble`. The LCD operates in 4-bit mode, therefore sending an 8bit command requires splitting the command into 2 4-bit parts. It then will perform a specific set of actions, as specified by the manufacture of the LCD. These actions consist of setting the RS, R/W, and ENABLE pins low, sending a nibble, turning ENABLE high, delaying, and turning ENABLE low, then repeating. It will then delay by either 37µs or 1.52ms depending on the command it was sent.

Figure 10. LCD_command() Flowchart [2]

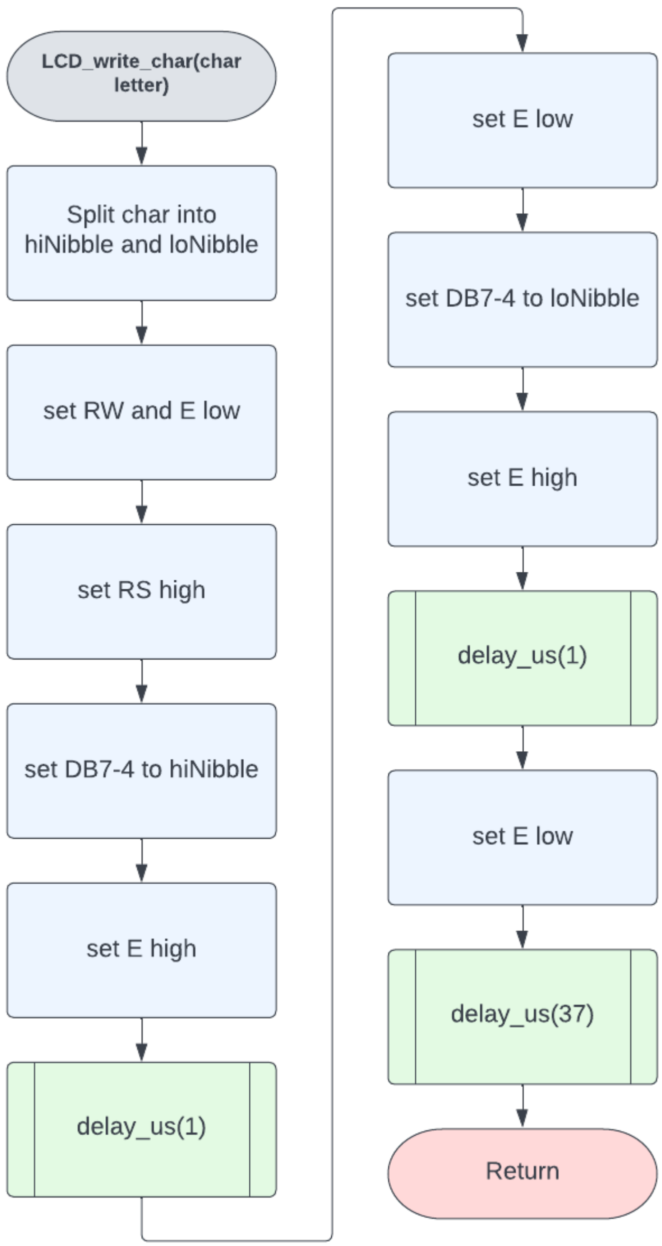


[2] Flowchart taken from EE 329 Canvas Course Page – created by Professor Paul Hummel

`void LCD_write_char(char letter)`

This function utilizes the same process as `LCD_command()` function, with the only difference being that RS will be high to allow for writing to the LCD and the delay at the end will be 37µs. It also has an input type of `char`, allowing for easy conversion to the standard ASCII values.

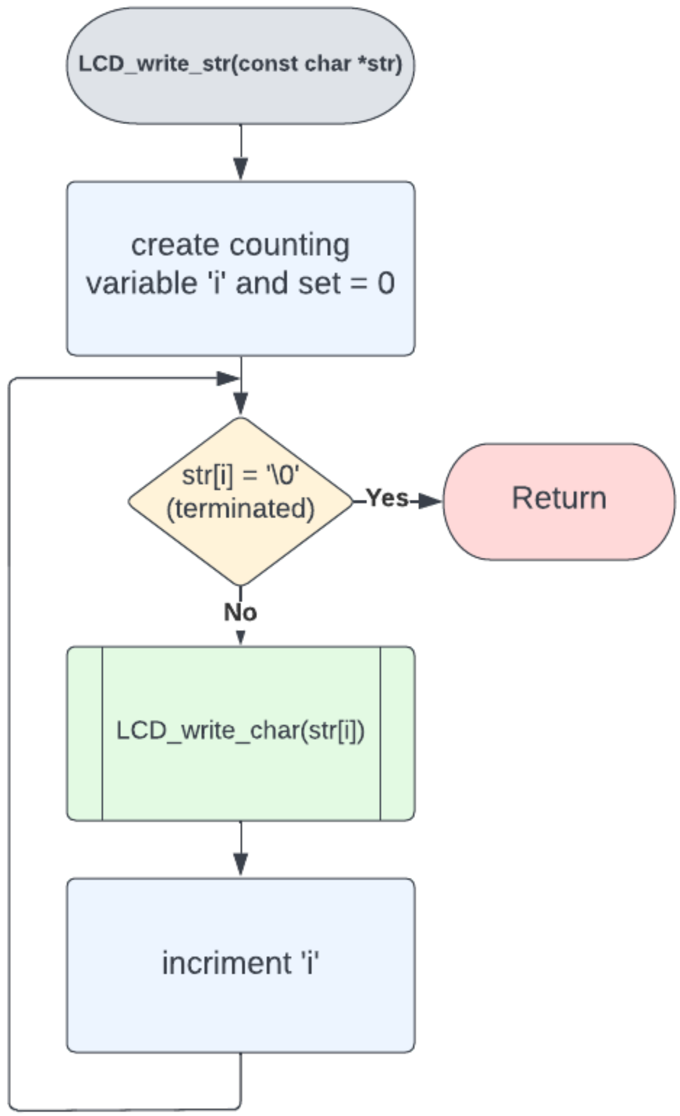
Figure 11. LCD_write_char() Flowchart



`void LCD write str(const char *str)`

This function utilizes a pointer type of data, basically a string of characters. By turning the input into a pointer, it will then be able to index from bit 0 to the termination of the string. The program automatically knows the string has terminated when it identifies the `char '\0'`. It will begin at bit 0, check if it is the termination character. If not it will print and increment the counter.

Figure 12. LCD_write_str() Flowchart

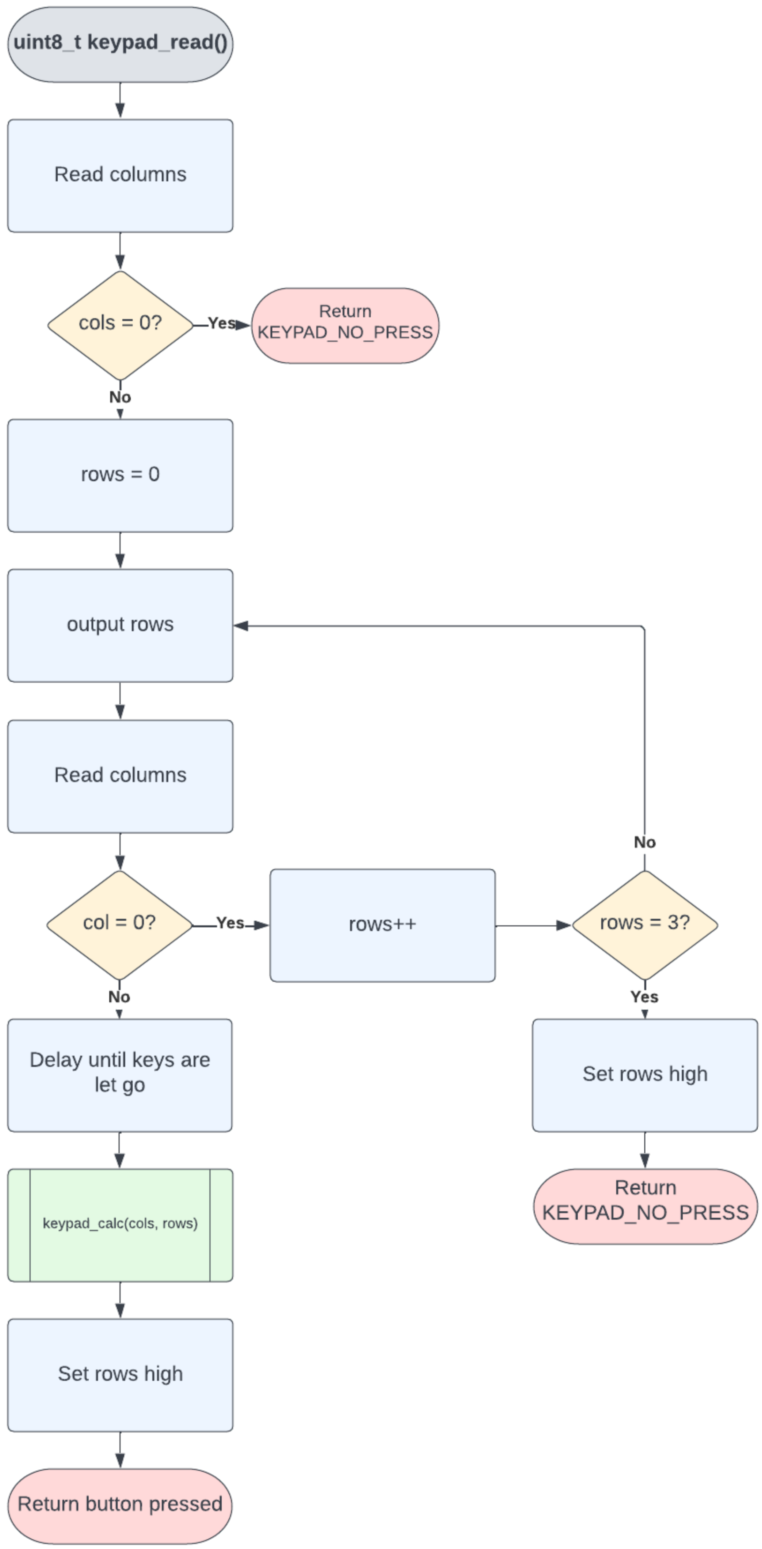


4.4.2. Keypad Functions

uint8_t keypad_read(void)

This function works by initializing variables `rows`, `cols`, `button`, and `row_output = 1`. These variables are used to keep track of what row and column is currently being incremented, what button is being pressed, and what value to output to the rows. After this initialization, the program reads the columns and checks if any of them are high. If one of these columns were to be high, that would mean that a button is being pressed because all rows were set high during the initialization. If no button was pressed, return an arbitrary very large value. However, if a button is pressed the program will then increment the rows one by one and will check if any columns are high as the rows are incremented. If a column is high, it will save the value of the rows and columns, then transfer those values into `uint8_t keypad_calc(uint8_t)`. After `uint8_t keypad_calc(uint8_t)` returns the value of the button press, it will save into the variable `button`. Lastly, the system will set all the rows high again and then return the value `button`.

Figure 13. keypad_read() Flowchart



`uint8 t keypad calc(uint8 t cols, uint8 t rows)`

This function works by taking the values `cols` and `rows` from the `keypad_read` function and using a math expression or if-statements to determine which value was pressed. If the rows are between 1-3 and columns are between 1-3, the system will use the following equation to determine what digit was pressed:

$$\text{key\_press} = \big((\text{rows} * 3) + (\text{cols} * 1) + 1\big)$$

(Eqn. 1)

If the column value is 4, then the key pressed must be a letter (because all letters are in column 4) and it will use the row value to determine the ASCII hex value corresponding to the letter with the following formula:

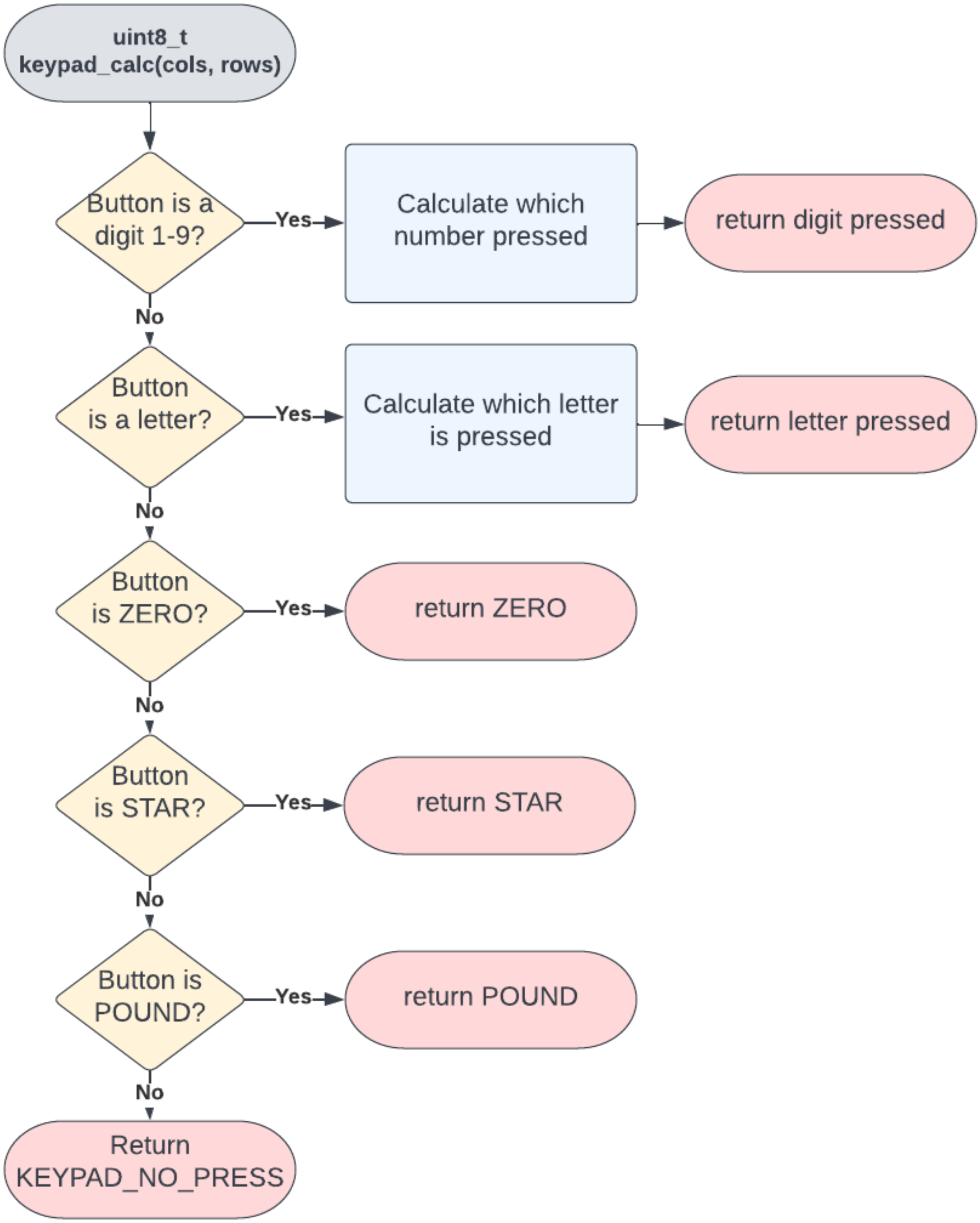
$$\text{key\_press} = \text{KEYPAD\_A} + \text{rows};$$

(Eqn. 2)

KEYPAD_A's hex value is equivalent to 0x41. If the row is 1, then the equation will add 1 to 0x41, obtaining 0x42 or the ASCII letter B. The same applies to the other characters.

Then, using if statements, the system will check if *, #, or 0 was pressed. If it cannot find any value it will return an arbitrary value identifying no key was pressed, otherwise it will return the value calculated.

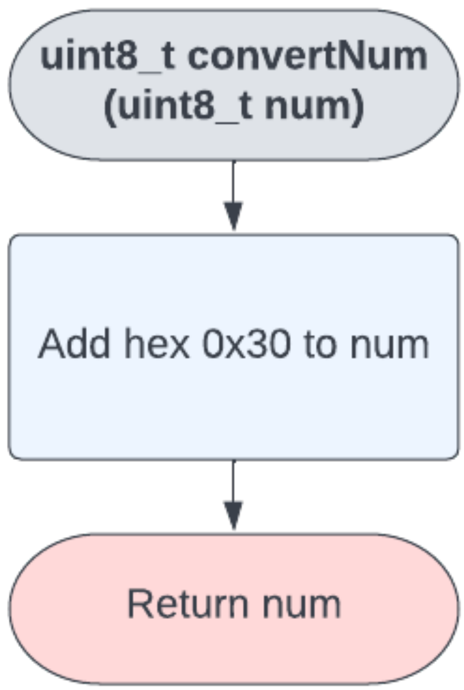
Figure 14. keypad_calc() Flowchart



`uint8_t convertNum(uint8_t num)`

This function simply adds hex 0x30 to the digit returned by `keypad_read`. The reason being is because zero is equivalent to 0x30, 1 is equivalent to 0x31, 2 is equivalent to 0x32, and so on. So to convert from a digit to an ASCII char, this is needed. The reason for this conversion is to save the digit in integer/decimal form to `pin` and then print the ASCII char.

Figure 15. convertNum() Flowchart

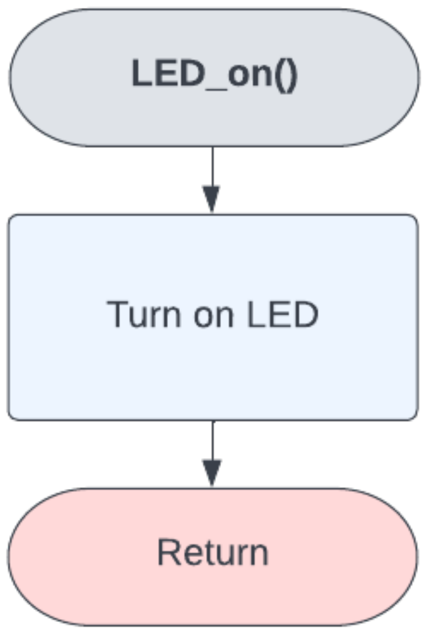


4.4.3. LED Functions

`void LED_on(void)`

This function simply outputs a value of 0x4000 to the GPIOB output data register. The value is equivalent to 0x4000 because the LED is attached to pin 15 and 0x4000 places a 1 in bit 15, therefore turning on the LED. This function is used to prevent having to repetitively type the commands for operating the LED.

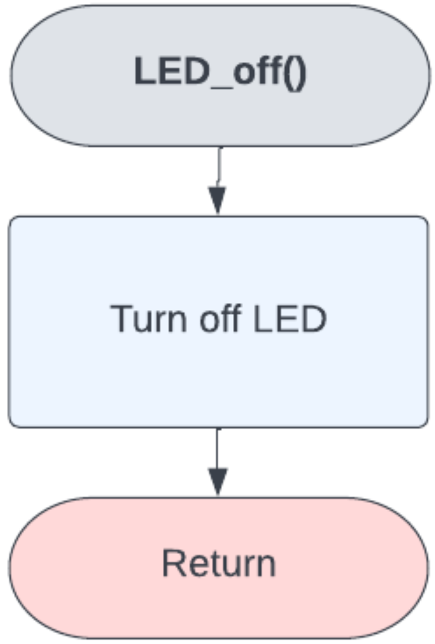
Figure 16. LED_on() Flowchart



`void LED_off(void)`

This functions works the exact same as the `void LED_on(void)` However it simply inverts the 0x4000 and &'s the output data register, effectively setting bit 15 to zero, turning off the LED.

Figure 17. LED_off() Flowchart



5. Appendices

Appendix A – References

[1] P. Hummel and J. Green, “STM32 Lab Manual,” *Google Docs*. [Online]. Available: https://docs.google.com/document/d/1NdE5188B2JWkEPdFAOdvxrEYo0R90Cg7wsG6oqHYOwk/edit#. [Accessed: 20-Apr-2022].

[2] P. Hummel, “LCD_Command_flowchart.pdf,” *Cal Poly Canvas*, 08-Apr-2022. [Online]. Available: https://canvas.calpoly.edu/courses/79417/files/7169933?module_item_id=2076255.

[3] “ST32L476 Data Sheet,” *ST.com*. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l476je.pdf. [Accessed: 20-Apr-2022].

[4] “ST32L476 Reference Manual,” *ST.com*. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. [Accessed: 20-Apr-2022].

[5] M. Tsoi, “Incomplete Guide to C (With A Focus on Embedded Systems Development),” 29-Mar-2021. .

[6] “NHD-0216HZ-FSW-FBW-33V3C Datasheet,” *Newhaven Display*. [Online]. Available: https://newhavendisplay.com/specs/NHD-0216HZ-FSW-FBW-33V3C.pdf. [Accessed: 20-Apr-2022].

Appendix B – Source Code

Source code is attached on the following pages:

```c
// EE 329 - 01
// Authors: Ethan Najmy
// Project #1 - Digital Lockbox

/* ----------- main.c ------------ */

// Include header files
#include "main.h"
#include "lcd.h"
#include "keypad.h"
#include "delay.h"
#include "led.h"

// Initialize local function
void SystemClock_Config(void);

int main(void) {
    /* Create the states and a new variable type called 'State_Type'
     * The states include ST_INIT, ST_LOCKED, ST_UNLOCKED, ST_WRONG, and ST_NEW_PIN
     * The states begin in ST_INIT
     */
    typedef enum {ST_INIT, ST_LOCKED, ST_UNLOCKED, ST_WRONG, ST_NEW_PIN} State_Type;
    State_Type state = ST_INIT;

    // Declare variables, DEFAULT_PIN = 1234 and can be changed
    uint8_t button, count;
    uint16_t DEFAULT_PIN = 1234, pin;

    // Initialize System Functions
    HAL_Init();
    SystemClock_Config();

    // Call functions to initialize LCD, keypad, and LED
    LCD_init();
    keypad_init();
    LED_init();

    // Enter FSM in infinite while loop
    while (1) {
        // Switch statement to jump between states
        switch(state) {

        // Inital State - runs on startup, resets system
        case ST_INIT:
            LCD_command(CLEAR_HOME);        // Clear LCD
            LED_on();                           // Turn on locking indicator
            LCD_write_str("SYSTEM LOCKED.");// Print
            LCD_command(NEW_LINE);              // New line to not cut off text
            LCD_write_str("ENTER PIN: ");       // Print
            state = ST_LOCKED;                  // Set state to ST_LOCKED
            pin = 0;                        // Reset pin variable
            count = 0;                          // Reset count variable
            break;                              // Break out of this state, will go to ST_LOCKED

        // Locked State
        case ST_LOCKED:
            // Read button continuously and wait for a press
            button = keypad_read();
            while (button == KEYPAD_NO_PRESS) {
                button = keypad_read();
            }

            // If star was pressed, reset system and clear screen
            if (button == KEYPAD_STAR) {
                state = ST_INIT;
                break;
            }

```

```c
            // If a digit was pressed print the digit
            if (button <= 9) {
                LCD_write_char(convertNum(button)); // Write the char after converting the to ASCII
                pin += button;                      // Add to pin with new button value
                count++;                        // Increment count
                if (count < 4)                  // Multiply if 3 or less digits have been entered
                    pin *= 10;                  // Allows for the int to be shifted hundreds places
            }

            // If pound was pressed and pin entered matches, UNLOCK
            // If pound was pressed and pin doesnt match, WRONG
            if (button == KEYPAD_POUND){
                if ((pin == DEFAULT_PIN) && (count >= 1)) {
                    state = ST_UNLOCKED;
                    break;
                } else {
                    state = ST_WRONG;
                    break;
                }
            }
            break;

        // UNLOCKED State - will shut off LED and print.
        case ST_UNLOCKED:
            LED_off();
            LCD_command(CLEAR_HOME);
            LCD_write_str("UNLOCKED");
            LCD_command(NEW_LINE);
            LCD_write_str("*-lock/#-new pin");
            button = keypad_read();

            // Wait for button to be pressed
            while (button == KEYPAD_NO_PRESS)
                button = keypad_read();

            // If the button pressed was star, lock back up
            if (button == KEYPAD_STAR) {
                state = ST_INIT;
                break;

            // If pound was pressed, go to NEW_PIN state
            } else if (button == KEYPAD_POUND){
                state = ST_NEW_PIN;
                break;
            // If nothing pressed, break and repeat
            } else {
                break;
            }

        // WRONG State - will print saying wrong pin and wait for #
        case ST_WRONG:
            LCD_command(CLEAR_HOME);
            LCD_write_str("WRONG PIN.");
            LCD_command(NEW_LINE);
            LCD_write_str("TRY AGAIN.");

            // Wait for # to be pressed so the user can acknowledge wrong pin
            button = keypad_read();
            while (button != KEYPAD_POUND){
                button = keypad_read();
            }
            // When # is pressed, go back to INIT state
            state = ST_INIT;
            break;

        // NEW_PIN State - user can enter new pin
        case ST_NEW_PIN:
            // Reset pin and count variables
```

```
            pin = 0;
            count = 0;

            // Clear LCD and print
            LCD_command(CLEAR_HOME);
            LCD_write_str("ENTER NEW 4");
            LCD_command(NEW_LINE);
            LCD_write_str("DIGIT PIN: ");

            // Same process as printing to LCD in ST_LOCKED
            while (1) {
                button = keypad_read();
                if (button <= 9) {
                    LCD_write_char(convertNum(button));
                    pin += button;
                    count++;
                    if (count < 4)
                        pin *= 10;
                }

                // If # is pressed and 4 digits have been entered,
                // set the entered pin as the default pin and go to INIT
                if ((button == KEYPAD_POUND) && (count == 4)) {
                    DEFAULT_PIN = pin;
                    state = ST_INIT;
                    break;
                }
                // If * pressed, reset NEW_PIN state and clear LCD
                if (button == KEYPAD_STAR)
                    break;
            }

        }

    }

}

void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_MSI;
  RCC_OscInitStruct.MSIState = RCC_MSI_ON;
  RCC_OscInitStruct.MSICalibrationValue = 0;
  RCC_OscInitStruct.MSIClockRange = RCC_MSIRANGE_6;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_MSI;
```

```
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
  {
    Error_Handler();
  }
}

void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

```c
/* ---------- delay.h ---------- */
#ifndef SRC_DELAY_H_
#define SRC_DELAY_H_

// Include function declarations / prototypes
void SysTick_Init(void);
void delay_us(const uint32_t);

#endif /* SRC_DELAY_H_ */

```

```c
/* ---------- delay.c ---------- */
#include "main.h"
#include "delay.h"

/* Configure SysTick Timer for use with delay_us function. This will break
 * break compatibility with HAL_delay() by disabling interrupts to allow for
 * shorter delay timing.
 */
void SysTick_Init(void){
    SysTick->CTRL |= (SysTick_CTRL_ENABLE_Msk |         // enable SysTick Timer
                      SysTick_CTRL_CLKSOURCE_Msk);      // select CPU clock
    SysTick->CTRL &= ~(SysTick_CTRL_TICKINT_Msk);       // disable interrupt,
                                                        // breaks HAL delay function
}

/* Delay function using the SysTick timer to count CPU clock cycles for more
 * precise delay timing. Passing a time of 0 will cause an error and result
 * in the maximum delay. Short delays are limited by the clock speed and will
 * often result in longer delay times than specified. @ 4MHz, a delay of 1us
 * will result in a delay of 10-15 us.
 */
void delay_us(const uint32_t time_us) {
    // set the counts for the specified delay
    SysTick->LOAD = (uint32_t)((time_us * (SystemCoreClock / 1000000)) - 1);
    SysTick->VAL = 0;                                       // clear the timer count
    SysTick->CTRL &= ~(SysTick_CTRL_COUNTFLAG_Msk);         // clear the count flag
    while (!(SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk)); // wait for the flag
}
```

```c
/* ----------- keypad.h ------------ */
#ifndef SRC_KEYPAD_H_
#define SRC_KEYPAD_H_

// Include #defines
#define ROW0 0x01       // Define ROW0 as 'pin 1'
#define ROW1 0x02       // Define ROW1 as 'pin 2'
#define ROW2 0x04       // Define ROW2 as 'pin 3'
#define ROW3 0x08       // Define ROW3 as 'pin 4'
#define COL0 0x10       // Define COL0 as 'pin 5'
#define COL1 0x20       // Define COL1 as 'pin 6'
#define COL2 0x40       // Define COL2 as 'pin 7'
#define COL3 0x80       // Define COL3 as 'pin 8'
#define KEYPAD_ROW_MASK (ROW0 | ROW1 | ROW2 | ROW3) // KEYPAD_ROW_MASK = 0x0F
                                                    // - when ANDed, only gives ROW values
#define KEYPAD_COL_MASK (COL0 | COL1 | COL2 | COL3) // KEYPAD_COL_MASK = 0xF0
                                                    // - when ANDed, only gives COL values
#define KEYPAD_NO_PRESS 0xF3// Return 0b11110011 ('?') when no key is pressed
#define KEYPAD GPIOA        // Define keypad as GPIOD for ease of switching ports

#define KEYPAD_0 0x00
#define KEYPAD_A 0x41
#define KEYPAD_B 0x42
#define KEYPAD_C 0x43
#define KEYPAD_D 0x44
#define KEYPAD_STAR 0x2A
#define KEYPAD_POUND 0x23

// Include function declarations / prototypes
void keypad_init(void);
uint8_t keypad_read(void);
uint8_t keypad_calc(uint8_t, uint8_t);
uint8_t convertNum(uint8_t);

#endif /* SRC_KEYPAD_H_ */

```

```c
/* ----------- keypad.c ------------ */
#include "main.h"
#include "keypad.h"

void keypad_init(void) {
  /* -- Configure GPIO D -- */
  RCC->AHB2ENR |= (RCC_AHB2ENR_GPIOAEN);//Enable GPIO D Clock

  KEYPAD->MODER &= ~(GPIO_MODER_MODE0       // Set MODE[0:7][1:0] to 0
          | GPIO_MODER_MODE1            // Will keep pins 4-7 as 0 for input mode
          | GPIO_MODER_MODE2
          | GPIO_MODER_MODE3
          | GPIO_MODER_MODE4
          | GPIO_MODER_MODE5
          | GPIO_MODER_MODE6
          | GPIO_MODER_MODE7);

  KEYPAD->MODER |= (GPIO_MODER_MODE0_0      // Set MODE[0:3][0] to 1
          | GPIO_MODER_MODE1_0              // Set as output pins
          | GPIO_MODER_MODE2_0
          | GPIO_MODER_MODE3_0);

  KEYPAD->OTYPER &= ~(GPIO_OTYPER_OT0       // Set OTYPE[0:3] to 0
          | GPIO_OTYPER_OT1                 // Set as push-pull
          | GPIO_OTYPER_OT2
          | GPIO_OTYPER_OT3);

  KEYPAD->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED0 // Set OSPEED[0:3] to 0
          | GPIO_OSPEEDR_OSPEED1        // Set output speed as low
          | GPIO_OSPEEDR_OSPEED2
          | GPIO_OSPEEDR_OSPEED3);

  KEYPAD->PUPDR &= ~(GPIO_PUPDR_PUPD0       // Set PUPD[0:7][1:0] to 0
          | GPIO_PUPDR_PUPD1            // Will keep pins 0-3 at 0
          | GPIO_PUPDR_PUPD2
          | GPIO_PUPDR_PUPD3
          | GPIO_PUPDR_PUPD4
          | GPIO_PUPDR_PUPD5
          | GPIO_PUPDR_PUPD6
          | GPIO_PUPDR_PUPD7);

  KEYPAD->PUPDR |= (GPIO_PUPDR_PUPD4_1      // Set PUPD[4:7][1] to 1
          | GPIO_PUPDR_PUPD5_1              // Enables pull-down resistors
          | GPIO_PUPDR_PUPD6_1
          | GPIO_PUPDR_PUPD7_1);

  KEYPAD->ODR &= ~(KEYPAD_ROW_MASK);        // Set rows to high
  KEYPAD->ODR |= (KEYPAD_ROW_MASK);
}

// READ KEYPAD FUNCTION
uint8_t keypad_read(void) {

    /* -- Initialize variables -- */

    // rows, cols - keep track of row # and column #
    // button - saves key pressed on keypad
    // row_output - enables rows to be turned high one at a time
    uint8_t rows, cols, button, row_output = 1;

    cols = KEYPAD->IDR & KEYPAD_COL_MASK;   // Read input register and only save columns
    if (cols == 0)                          // If no keys pressed (cols = 0), return KEYPAD_NO_PRESS
        return KEYPAD_NO_PRESS;

    /* -- Row Incrementer -- */
    for(rows = 0; rows < 4; rows++) {
        KEYPAD->ODR &= ~(KEYPAD_ROW_MASK);
        KEYPAD->ODR |= row_output;          // Set ROW0 high
```

```
        row_output *= 2;                                // Multiply row by 2, output to ROW1,2,3
        cols = (KEYPAD->IDR & KEYPAD_COL_MASK);             // Read columns
        if (cols != 0) {                                // If no column high, skip loop

            while (KEYPAD->IDR & KEYPAD_COL_MASK){
                //  Delay while buttons are pressed
            }

            button = keypad_calc(cols,rows);            // Calc key press from row and col value
                                                            // and save to button
            KEYPAD->ODR &= ~(KEYPAD_ROW_MASK);              // Set rows high
            KEYPAD->ODR |= (KEYPAD_ROW_MASK);
            return button;                                  // Return button


        }

    }
    KEYPAD->ODR &= ~(KEYPAD_ROW_MASK);  // Set rows high
    KEYPAD->ODR |= (KEYPAD_ROW_MASK);
    return KEYPAD_NO_PRESS;                         // Return 0xFF
}

// KEYPAD CALCULATE WHAT BUTTON WAS PRESSED
uint8_t keypad_calc(uint8_t cols, uint8_t rows) {
    uint8_t key_press;                              // Initialize variable to save key value

    cols = (cols >> 5);                             // Shift column value 5 bits, allows columns to
                                                    // be numbered 0,1,2,4

    if ((cols < 4) && (rows < 3)) {                 // 4x4 KEYPAD, check IF value is number 1-9
        key_press = ((rows*3)+(cols*1)+ 1);         // Use eqn to calculate number key pressed and
                                                    // add 0x30 to convert to ASCII value

    } else if (cols == 4) {                         // If a value in column 4 was pressed
        key_press = KEYPAD_A + rows;            // Add rows to KEYPAD_A (10),
                                                    // if row = 0, key_press = 10 (A),
                                                    // if row = 1, key_press = 11 (B), etc.

    } else if (rows == 3 && cols == 0) {        // If bottom left button, key pressed = *
        key_press = KEYPAD_STAR;

    } else if (rows == 3 && cols == 1) {        // If bottom middle button, key pressed = 0
        key_press = KEYPAD_0;

    } else if (rows == 3 && cols == 2) {        // If bottom right button, key pressed = #
        key_press = KEYPAD_POUND;

    } else
        key_press = KEYPAD_NO_PRESS;            // If nothing pressed, return 0xFF

    return key_press;                               // Return key_press value
}

// USED TO CONVERT DIGIT 0-9 TO ASCII CHAR VALUE
uint8_t convertNum (uint8_t num) {
    num += 0x30;    // Add hex 0x30 and save as num.
    return num;         // Return num
}

```

```c
/* ---------- lcd.h ---------- */

#ifndef SRC_LCD_H_
#define SRC_LCD_H_

/* -- GPIO Ports -- */
// These #defines are used to easily change GPIO pins if needed
#define LCDD GPIOD  // Allows easy change of port for LCD if needed
#define RS  0x01// RS is on GPIO PD0, therefore gets bit 1
#define RW  0x02// RW is on GPIO PD1, therefore gets bit 2
#define E   0x04// E is on GPIO PD2, therefore gets bit 3
#define DB4 0x10// LCD DB4 is on GPIO PD4, therefore gets bit 5
#define DB5 0x20// LCD DB5 is on GPIO PD5, therefore gets bit 6
#define DB6 0x40// LCD DB6 is on GPIO PD6, therefore gets bit 7
#define DB7 0x80// LCD DB7 is on GPIO PD7, therefore gets bit 8

/* -- Masks -- */
// These masks are used to only modify desired bits
#define DB_MASK (DB4 | DB5 | DB6 | DB7) // Allows masking of DB ports
#define RS_RW_E_MASK (RS | RW | E)      // Allows masking of RS, RW, & E
#define E_MASK 0x04                     // Allows masking of only E

/* -- Commands -- */
// These #defines correspond to hex values used by the ST7066U to process commands
#define NEW_LINE 0xC0                   // 0b11000000 - new line command
#define CLEAR_HOME 0x01                 // 0b00000001 - clear & go home command
#define DISPLAY_ON 0x0F                 // 0b00001111 - turn display on
#define ENTRY_MODE_SET 0x06             // 0b00000110 - set entry mode

/* -- Function declarations / prototypes -- */
void LCD_init(void);                // Initialize LCD
void LCD_command(uint8_t);                  // Send LCD a single 8-bit command
void LCD_write_char(char);                  // Write a character to the LCD
void LCD_write_str(const char *str);    // Write a string to the LCD
void LCD_write_n_ctr_str(const char *str);  // Write and center a string to LCD

#endif
```

```c
/* ---------- lcd.c ---------- */
#include "main.h"
#include "lcd.h"
#include "delay.h"

void LCD_init(void) {                       // Initialize LCD function

    /* -- Configure GPIO D -- */
    RCC->AHB2ENR |= (RCC_AHB2ENR_GPIODEN);  // Enable GPIO D Clock

    LCDD->MODER &= ~(GPIO_MODER_MODE0       // Mask MODER[0:2,4:7]
            | GPIO_MODER_MODE1
            | GPIO_MODER_MODE2
            | GPIO_MODER_MODE4
            | GPIO_MODER_MODE5
            | GPIO_MODER_MODE6
            | GPIO_MODER_MODE7);

    LCDD->MODER |= (GPIO_MODER_MODE0_0      // Set MODER[0:2,4:7][1] to one (output mode)
            | GPIO_MODER_MODE1_0
            | GPIO_MODER_MODE2_0
            | GPIO_MODER_MODE4_0
            | GPIO_MODER_MODE5_0
            | GPIO_MODER_MODE6_0
            | GPIO_MODER_MODE7_0);

    LCDD->OTYPER &= ~(GPIO_OTYPER_OT0       // Set OTYPER[0:2,4:7] to zero (push-pull)
            | GPIO_OTYPER_OT1
            | GPIO_OTYPER_OT2
            | GPIO_OTYPER_OT4
            | GPIO_OTYPER_OT5
            | GPIO_OTYPER_OT6
            | GPIO_OTYPER_OT7);

    LCDD->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED0 // Set OSPEEDR[0:2,4:7] to zero (low speed)
            | GPIO_OSPEEDR_OSPEED1
            | GPIO_OSPEEDR_OSPEED2
            | GPIO_OSPEEDR_OSPEED4
            | GPIO_OSPEEDR_OSPEED5
            | GPIO_OSPEEDR_OSPEED6
            | GPIO_OSPEEDR_OSPEED7);

    LCDD->PUPDR &= ~(GPIO_PUPDR_PUPD0       // Set PUPDR[O:2,4:7] to zero (no PU/PD resistor)
            | GPIO_PUPDR_PUPD1
            | GPIO_PUPDR_PUPD2
            | GPIO_PUPDR_PUPD4
            | GPIO_PUPDR_PUPD5
            | GPIO_PUPDR_PUPD6
            | GPIO_PUPDR_PUPD7);

    /* -- Begin LCD Initialization Process -- */
    SysTick_Init();

    delay_us(40000);                    // Initial 40ms delay
    LCD_command(0x03);                      // Set RS & R/W to zero (PD0 & PD1)
    delay_us(37);                           // Delay 37 us
    LCD_command(0x28);                      // Set font and line number
    delay_us(37);                           // Delay 37 us
    LCD_command(0x28);                      // Set font and line number, again
    delay_us(37);                           // Delay 37 us
    LCD_command(DISPLAY_ON);            // Turn display, cursor, and blink on
    delay_us(37);                           // Delay 37 us
    LCD_command(CLEAR_HOME);            // Clear display, return home
    delay_us(1520);                         // Delay 1.52 ms
    LCD_command(ENTRY_MODE_SET);        // Set mode to increment
}

void LCD_command(uint8_t command) {         // Send LCD a single 8-bit command
```

```c
    uint8_t hiNibble, loNibble;             // Initialize variables for upper 4 bits and lower 4 bits

    SysTick_Init();                         // Initialize delay function

    hiNibble = (command & 0xF0);        // Split command into hiNibble
    loNibble = ((command & 0x0F) << 4);     // Split command into loNibble
        /* Bits 4-7 are used (in F0 and with the shift left by 4)
         * due to use of GPIO PD4-7, thus making outputting easier
         */

    LCDD->ODR &= ~(RS_RW_E_MASK);           // Mask RS, R/W, and E to zero (PD0, PD1, PD2)
    LCDD->ODR &= ~(DB_MASK);            // Mask DB4-7 to zero (PD4-7)
    LCDD->ODR |= hiNibble;                  // Set DB4-7 to hiNibble
    LCDD->ODR &= ~(E_MASK);                 // Mask E to zero (PD2)
    LCDD->ODR |= E;                         // Set E to one
    delay_us(1);                        // Delay > 460ns
    LCDD->ODR &= ~(E_MASK);                 // Mask E to zero
    LCDD->ODR &= ~(DB_MASK);            // Mask DB4-7 to zero
    LCDD->ODR |= loNibble;                  // Set DB4-7 to loNibble
    LCDD->ODR &= ~(E_MASK);                 // Mask E to zero
    LCDD->ODR |= E;                         // Set E to one
    delay_us(1);                        // Delay > 460ns
    LCDD->ODR &= ~(E_MASK);                 // Mask E to zero

    /* If statement to check timing, only two commands have timing
     * different than 37 us, and those two commands have a HEX value
     * equal or less than 2 */
    if (command > 0x02){
        delay_us(37);       // Delay by 37 us
    } else {
        delay_us(1520);     // Delay by 1.52 ms
    }
}

void LCD_write_char(char letter) {      // Write a character to the LCD
    uint8_t hiNibble, loNibble;         // Initialize variables for upper and lower 4 bits

    SysTick_Init();                     // Initialize delay function

    hiNibble = (letter & 0xF0);         // Split command into hiNibble
    loNibble = ((letter & 0x0F) << 4);  // Split command into loNibble

    LCDD->ODR &= ~(RS_RW_E_MASK);       // Mask RS, RW, and E to zero
    LCDD->ODR |= (RS);                  // Set RS to 1, allowing for writing
    LCDD->ODR &= ~(DB_MASK);        // Mask only data pins to zero
    LCDD->ODR |= hiNibble;              // Set only data pins to hiNibble
    LCDD->ODR &= ~(E_MASK);             // Mask enable (E) to zero
    LCDD->ODR |= E;                     // Set E high
    delay_us(1);                    // Delay > 460ns
    LCDD->ODR &= ~(E_MASK);             // Set E to zero
    LCDD->ODR &= ~(DB_MASK);        // Mask only data pins to zero
    LCDD->ODR |= loNibble;              // Set only data pins to loNibble
    LCDD->ODR &= ~(E_MASK);             // Mask E to zero
    LCDD->ODR |= E;                     // Set E to one
    delay_us(1);                    // Delay > 460ns
    LCDD->ODR &= ~(E_MASK);             // Set E to zero
    delay_us(37);                       // Delay 37 ns
}

void LCD_write_str(const char *str){
    uint8_t i = 0;              // Initialize variables

    /* The following while statement indexes through the string that was inputed
     * starting at bit 0, printing each character until the index is greater than
     * the number of characters found to be on the first line, it will then move
     * to line 2 and begin printing the rest of the characters.
     */
    while (str[i] != '\0'){
```

```
        LCD_write_char(str[i++]);
    }
}

// THIS FUNCTION IS MEANT FOR PERSONAL USE AND NOT MEANT FOR GRADING / SUBMISSION
void LCD_write_n_ctr_str(const char *str){  // Function to write and center a string to LCD


```

```c
/* ---------- led.h ---------- */

#ifndef SRC_LED_H_
#define SRC_LED_H_


// Include Pound Defines
#define LED GPIOB   // Define LED as GPIOB
#define LED3 0x4000 // Define LED3 as Pin 15

// Include function definitions / prototypes
void LED_init(void);
void LED_on(void);
void LED_off(void);

#endif

```

```c
/* ---------- led.c ---------- */

#include "main.h"
#include "led.h"

void LED_init(void) {

    // Initialize GPIOB and PIN 14
    RCC->AHB2ENR |= (RCC_AHB2ENR_GPIOBEN);

    // Set pin 14 to output mode
    LED->MODER &= ~(GPIO_MODER_MODE14);
    LED->MODER |= (GPIO_MODER_MODE14_0);

    // Set pin 14 to PP
    LED->OTYPER &= ~(GPIO_OTYPER_OT14);

    // Set pin 14 to low speed
    LED->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED14);

    // Set pin 14 to no PUPD
    LED->PUPDR &= ~(GPIO_PUPDR_PUPD14);
}

// Turn LED on
void LED_on(void) {
    // LED3 has hex value 0x4000, giving a 1 at bit 15
    LED->ODR &= ~(LED3);
    LED->ODR |= (LED3);
}

// Turn LED off
void LED_off(void) {
    LED->ODR &= ~(LED3);
}

```