Project 3 - 8x8 LED Draw

Cal Poly, San Luis Obispo

CPE 316-01-2234

Paul Hummel

Noah Masten

Spring Quarter

June 8, 2023

Device Behavior

This project consists of a terminal-based application that allows the user to “draw” on an 8x8 ASCII grid. Whatever color the user draws on the grid is reflected on an 8x8 LED Matrix, allowing the user to essentially “draw” with LEDs. The project utilizes an STM32L476RGT3 microcontroller board and a WS2812B 8x8 LED Flex Panel. To write data to the LEDs, the program uses Direct Memory Access (DMA) with Pulse Width Modulation (PWM). The user interface is composed of ASCII characters, transmitted via UART.

System Specifications

Table 1. System Specifications Table

8x8 LED Draw	
Drawing Colors	Red, Orange, Yellow, Green, Blue, Indigo, Violet, Pink, Brown, White
Usable Keys (Home Screen)	W, A, S, D, C, Space, E, R
Keyboard Controls (Home Screen)	**W:** Cursor Up **A:** Cursor Left **S:** Cursor Down **D:** Cursor Right **C:** Select Color **Space:** Draw Pixel **E:** Erase Pixel **R:** Reset LEDS
Usable Keys (Color Selection)	R, O, Y, G, B, I, V, P, N, W
Keyboard Controls (Color Selection)	**R:** Red **O:** Orange **Y:** Yellow **G:** Green **B:** Blue **I:** Indigo **V:** Violet **P:** Pink **N:** Brown **W:** White
STM32L476RG Board	
Operating voltage	-0.3–4.0 V (min. – max.)
Total current draw	150 mA (max.)
Power consumption	0.6 W
Power connection	USB to Mini-B cable
WS2812B 8x8 LED Flex Panel	
LED count	64
Operating Voltage	5 V
Current Draw	50 mA/LED (max.) 3.2 A (max.), all 64 LEDs at full brightness

Terminal	
Baud Rate	115200 kbps

System Schematic

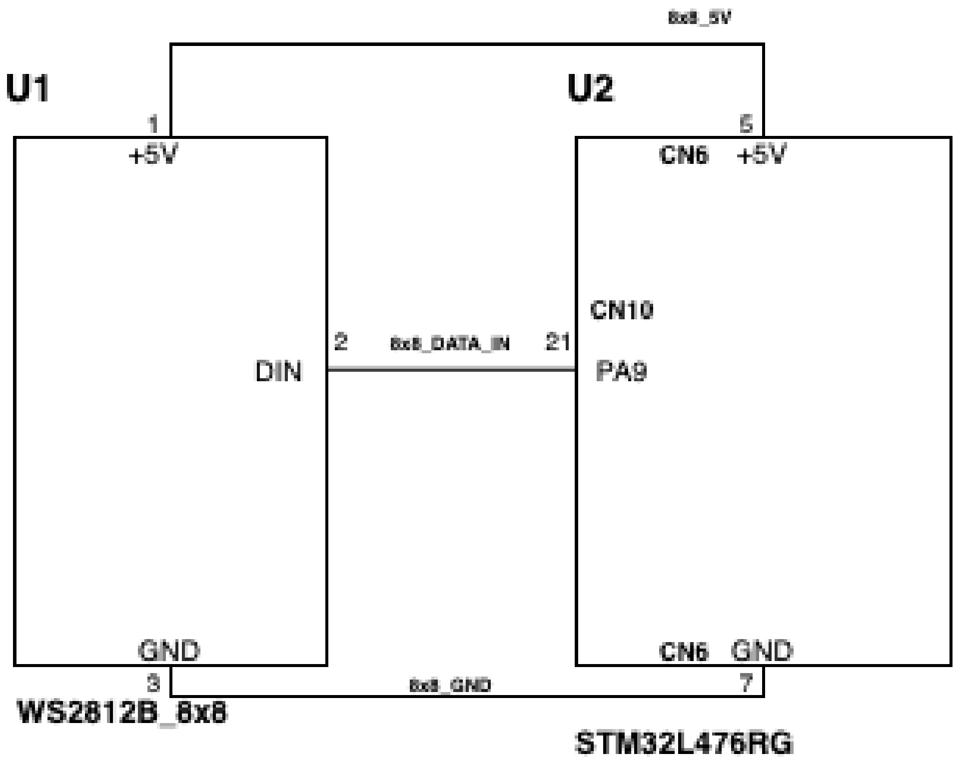

Figure 1. System Schematic

Software Architecture

1. Overview

The program starts by initializing all global variables, #define variables, FSM States, and color variables. Then, we initialize pin PA9 for GPIO, and Timer 1 Channel 2 for PWM and DMA.

The main loop of this program involves a Finite State Machine with six states:

- ST_DEFAULT
- ST_MOVE_CURSOR
- ST_SEL_COLOR
- ST_DRAW
- ST_ERASE
- ST_CLEAR

The program first enters ST_DEFAULT, which prints the home screen. The program waits for the user to press a key on the keyboard. If the user's keypress matches one of the keyboard controls, the program will enter one of the following states:

- **W, A, S, D** - ST_MOVE_CURSOR
- **C** - ST_SEL_COLOR
- **Space** - ST_DRAW
- **E** - ST_ERASE
- **R** - ST_CLEAR

Once the program has executed the function of the respective state, it returns to ST_DEFAULT, printing the home screen and waiting for the user to press another key.

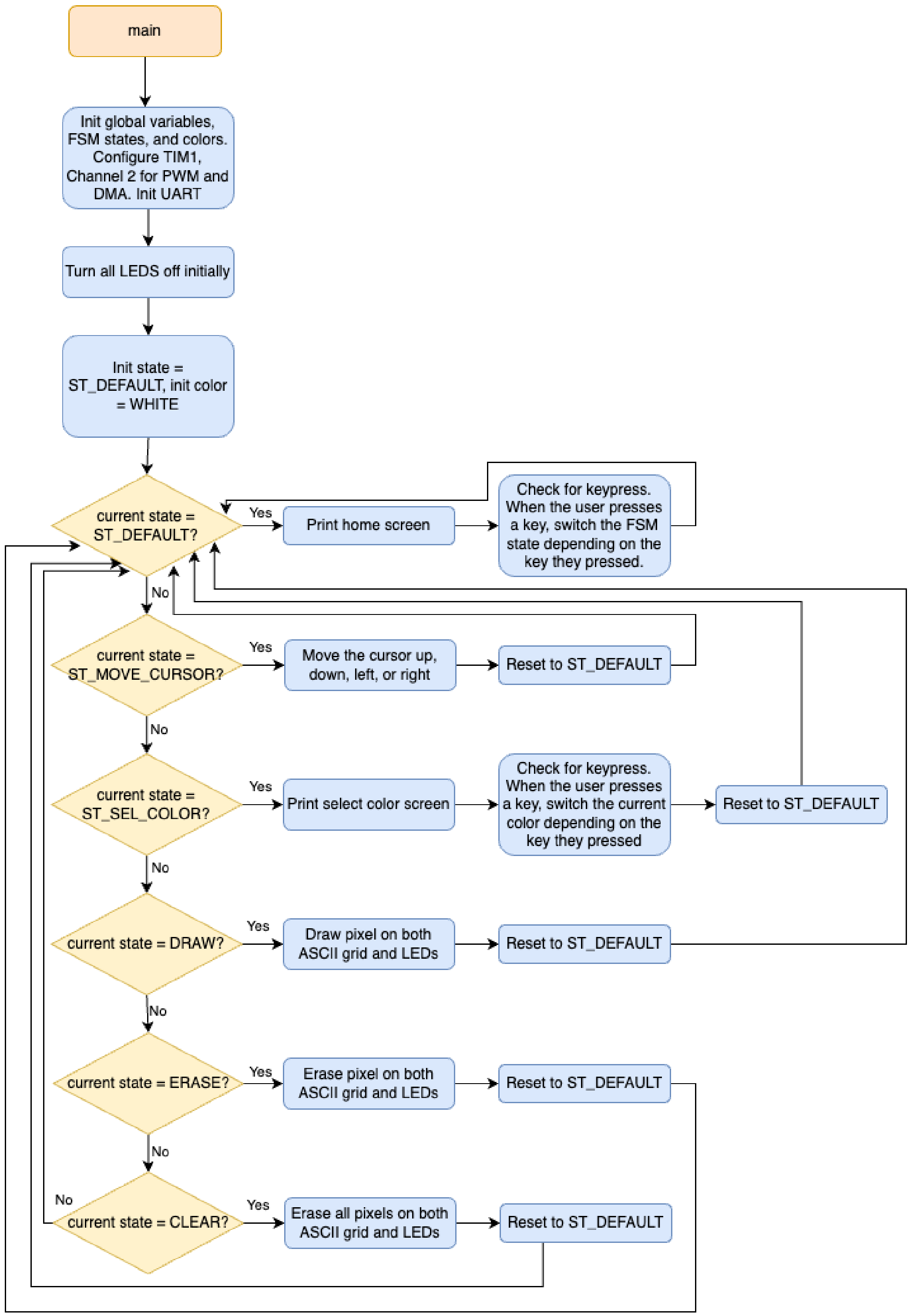

Figure 2. main() flowchart

2. Sending the Data

I used this source from ControllersTech.com [4], as well as the WS2812B datasheet [5] to figure out how to write data to the LEDs.

This section of the datasheet shows the transfer times of sending 1's and 0's to the LEDs:

Data transfer time(TH+TL=1.25µs±600ns)

T0H	0 code ,high voltage time	0.4us	±150ns
T1H	1 code ,high voltage time	0.8us	±150ns
T0L	0 code , low voltage time	0.85us	±150ns
T1L	1 code ,low voltage time	0.45us	±150ns
RES	low voltage time	Above 50µs	

Figure 3. Data transfer time of WS2812B [5]

Since the total transfer time is 1.25 µs, then we can calculate the frequency of the data transfer:

$$f = \frac{1}{1.25*10^{-6}} = 800\ kHz$$

Using a 72 MHz clock, we can calculate the period of our timer to achieve an 800 kHz frequency:

$$P = \frac{72*10^{6}}{800*10^{3}} = 90$$

The datasheet shows that sending a '1' requires the pulse to be high for 0.8 µs, and that sending a '0' requires the pulse to be high for 0.4 µs. This means that sending a '1' takes up 68% of the total transfer time ($\frac{0.8}{1.25}$), and that sending a '0' takes up 32% of the total transfer time ($\frac{0.4}{1.25}$). We can translate this to our recently calculated period:

$$90\ *\ 68\% = 61.2$$

$$90\ *\ 32\% = 28.8$$

So, anytime we want to write a ‘1’ to the LEDs, we send 61.2 through our PWM and DMA function, and anytime we want to write a ‘0’, we send 28.8. Because the timing for sending the data is relatively generous (±600 ns), values of 60 and 30 are used in the code. Additionally, there is a 50 μs reset code (shown as RES in Figure 6) that must take place between data transfers to indicate that a data transfer is finished. This is taken care of in the provided code.

Another thing to note is that the LEDs require the color data to be sent in the order of green, red, blue (8 bits each for values 0-255).

This program utilizes TIM1 Channel 2 on the STM32, initializing the timer for PWM and DMA. I did not end up writing this code myself, instead using the project configuration file to generate the code.

Here is a brief overview of the functions used from ControllersTech.com [5]:

- **Set_LED:** Stores the LED number and 8-bit red, green, blue values in an array.
- **Set_Brightness:** Sets the brightness of all LEDs (brightness values 0-45).
- **Reset_LED:** Sets the color of all LEDs to RGB value of (0, 0, 0), turning them all off.
- **WS2812_Send:** Iterates through all LED data (LED number, brightness, red value, green value, blue value) and sends either 60 (if color bit is a 1) or 30 (if color bit is a 0) through TIM1 PWM/DMA. After the DMA transfer is initiated, the function waits for the transfer to finish using the data transfer flag.
- **HAL_TIM_PWM_PulseFinishedCallback:** This function is called when a DMA pulse is finished, and stops the DMA transfer. Additionally, it sets the data transfer flag high, which is utilized in **WS2812_Send**.

3. Variables

Globals

Global variables are used to keep track of the current color, xy coordinates, and the key the user pressed. Also, I created a key pressed flag that indicates whether the user has pressed a key or not, used in both **main** and in the **USART2_IRQHandler**.

Color

To represent colors, I used the **typedef struct** keyword to create my own 'color' variable. The members of this struct are:

- **char name[10]:** The name of the color.
- **int red:** The RGB red value of the color.
- **int green:** The RGB green value of the color.
- **int blue:** The RGB blue value of the color.
- **char key:** The key press associated with the color on the Select Color screen (used for replacing characters on the ASCII grid).

This made it very easy to initialize the colors:

```
color RED = { "RED", 255, 0, 0, 'R' };

color ORANGE = { "ORANGE", 255, 127, 0, 'O' };

...
```

When the user changes colors, the global color variable is changed to one of the initialized colors.

Arrays

Two 8x8 global arrays are used during program execution. The first is **char UI_matrix**, which represents the ASCII grid in the home screen interface. This array is modified when the user draws or erases a pixel, changing the value at the indices of the cursor position.

The second array is **uint8_t LED_Matrix**, which represents the numerical values of the LEDs. Since the LED numbers do not conveniently ascend by rows and columns, I had to create an array where I could easily reference the LED number with xy coordinates. The LED numbers are arranged in a "snake" like formation:

```
uint8_t LED_Matrix[MATRIX_WIDTH][MATRIX_HEIGHT] = {

        { 0, 15, 16, 31, 32, 47, 48, 63 },
```

```
        { 1, 14, 17, 30, 33, 46, 49, 62 },
        { 2, 13, 18, 29, 34, 45, 50, 61 },
        { 3, 12, 19, 28, 35, 44, 51, 60 },
        { 4, 11, 20, 27, 36, 43, 52, 59 },
        { 5, 10, 21, 26, 37, 42, 53, 58 },
        { 6, 9,  22, 25, 38, 41, 54, 57 },
        { 7, 8,  23, 24, 39, 40, 55, 56 }
};
```

4. Finite State Machine

The states of the FSM were chosen based on the different functions of the key presses. Since the key presses can either move the cursor, select a color, draw, erase, or reset the pixels, the states were chosen accordingly. The state machine starts at ST_DEFAULT, and once a key is pressed, the program switches to the respective state, executes the function of that keypress, then returns to ST_DEFAULT. This was especially convenient since ST_DEFAULT prints the home screen, and we can easily re-print the home screen with our new cursor position, color, and ASCII grid modifications.

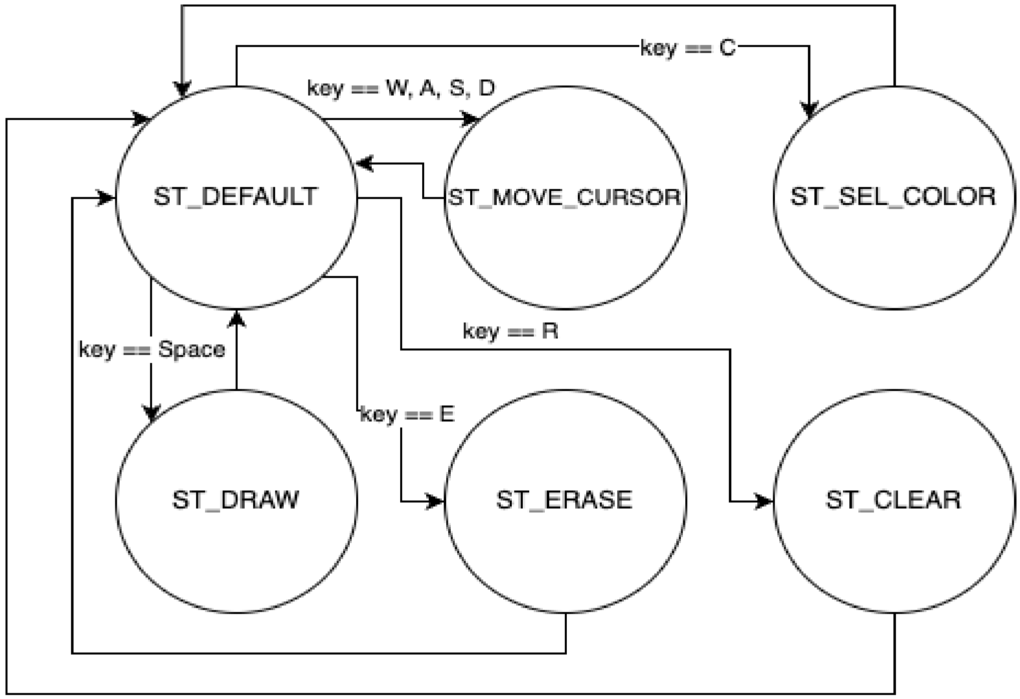

Figure 4. State Diagram for Finite State Machine

ST_DEFAULT

This state serves as the initial state, where the home screen is printed with the current cursor position, color, xy coordinates and ASCII grid modifications.

The state begins by printing the title in blue and bold text using UART escape codes, and the controls section in regular text. Then, the current color is printed using the current color variable, as well as the current x and y coordinates of the cursor. Under the xy coordinates, we print the ASCII grid with the color modifications and cursor position.

The program then waits for the user to press a key, using the key pressed flag, then resets the key pressed flag. Then the program changes the value of the current state, based on the key pressed.

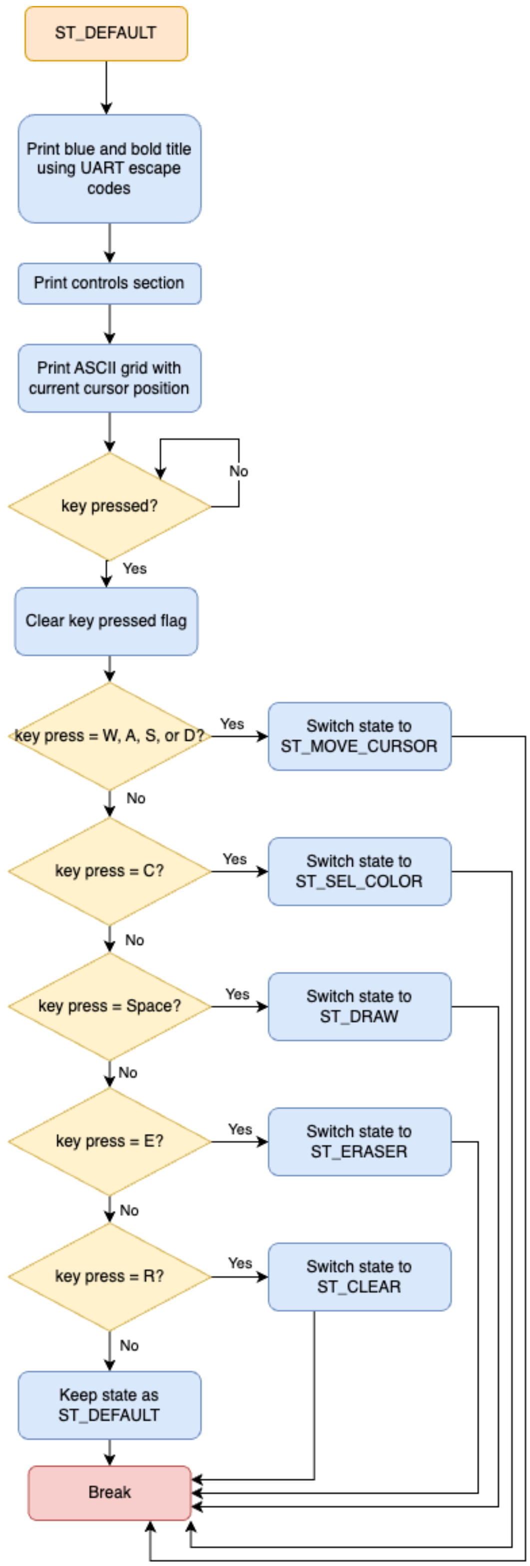

Figure 5. *ST_DEFAULT* flowchart

ST_MOVE_CURSOR

In this state, the x and y coordinates are either incremented or decremented, based on the key pressed. Based on the coordinate system I chose ([1,1] as top left and [8,8] as bottom right), 'W' decrements the y coordinate, 'A' decrements the x coordinate, 'S' increments the y coordinate, and 'D' increments the x coordinate. Finally, we change the current state to ST_DEFAULT in order to re-print the ASCII grid with the updated cursor position.

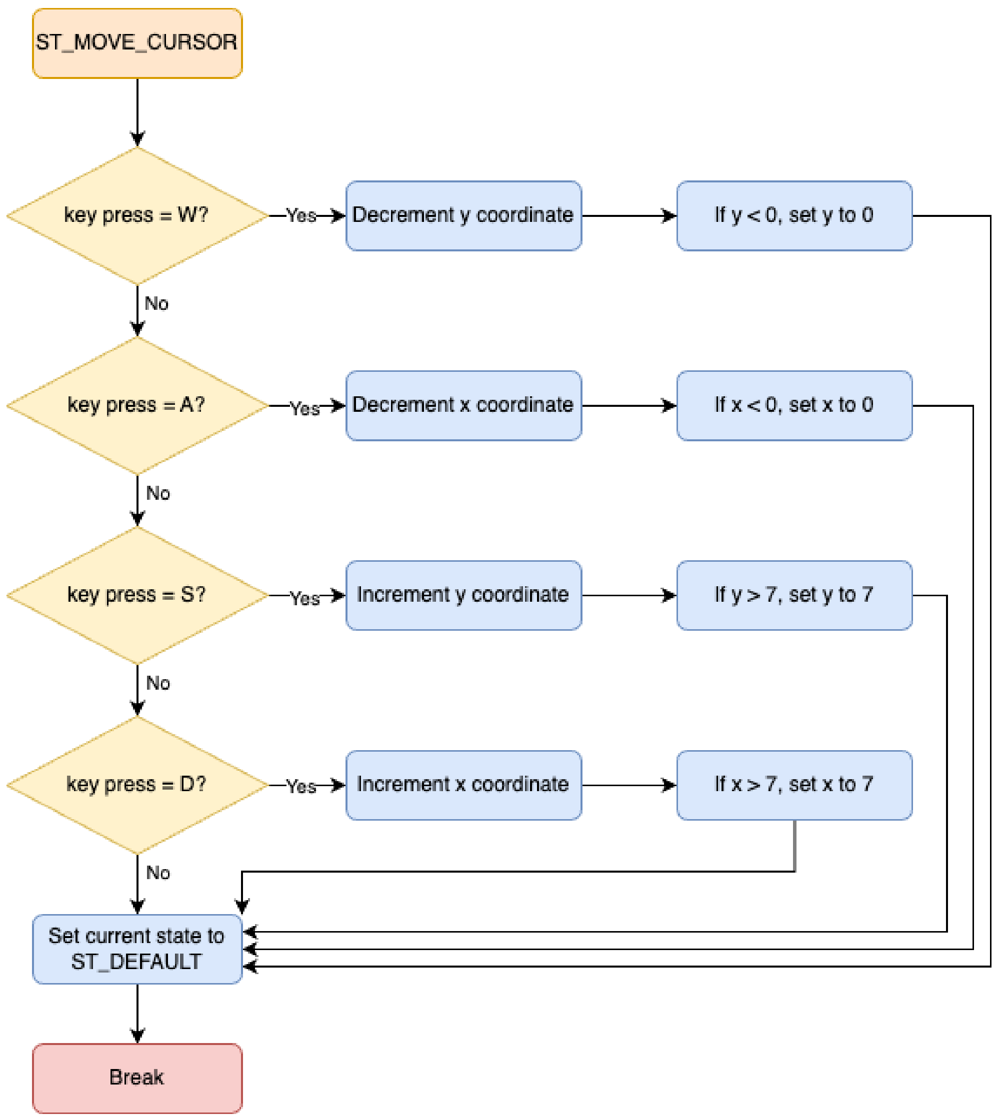

Figure 6. *ST_MOVE_CURSOR* flowchart

ST_SEL_COLOR

In this state, the interface changes from the home screen interface to the select color interface. This involves clearing the terminal screen and printing the blinking “Select Color” prompt and the color selection controls with UART. Similar to ST_DEFAULT, the program waits for the user to press a key, then clears the key pressed flag. The current color variable then changes to the color the user selects, based on the key press. Finally, we change the current state to ST_DEFAULT in order to re-print the home screen with the updated color selection.

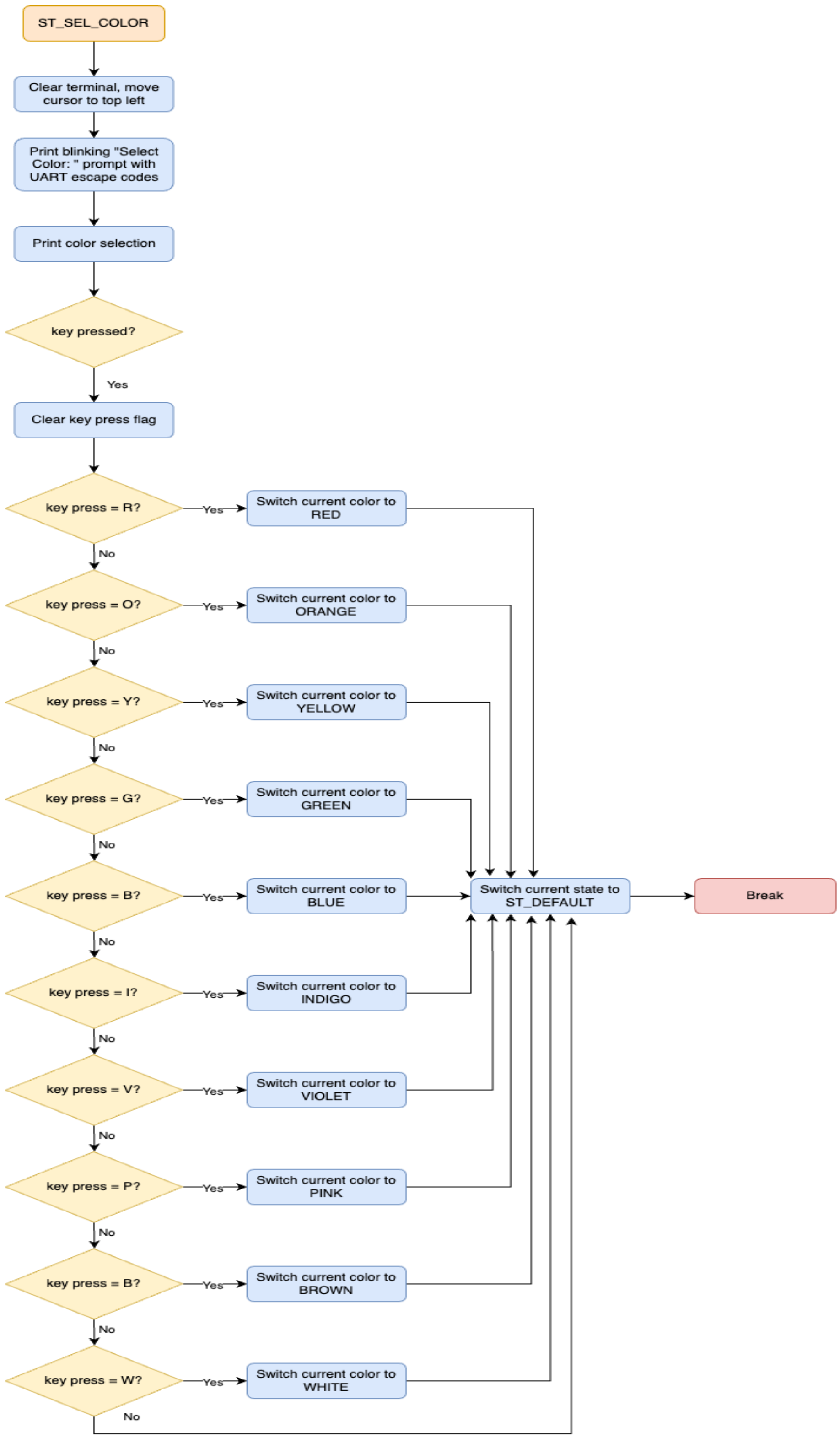

Figure 7. ST_SEL_COLOR flowchart

ST_DRAW

In this state, the color data is sent to the LEDs using our **Set_LED**, **Set_Brightness**, and **WS2812_Send** function. The brightness is hard-coded to a value of 2 (anything higher is too bright for my eyes). Additionally, the ASCII grid character at the indices of the current cursor position (x, y) is changed to the key press of the current color. Finally, we change the current state to ST_DEFAULT in order to re-print the modified ASCII grid.

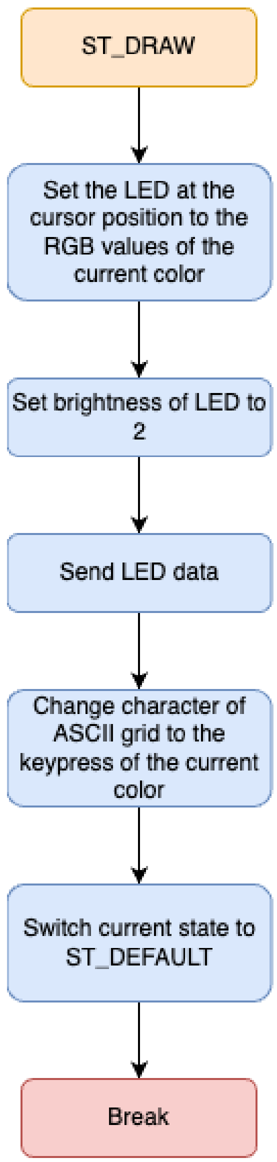

Figure 8. ST_DRAW flowchart

ST_ERASE

This state is functionally identical to ST_DRAW, except we use RGB values of (0, 0, 0) as arguments to our **Set_LED** function. The ASCII grid character at the indices of the cursor position is instead changed back to an asterisk. We change the current state to ST_DEFAULT.

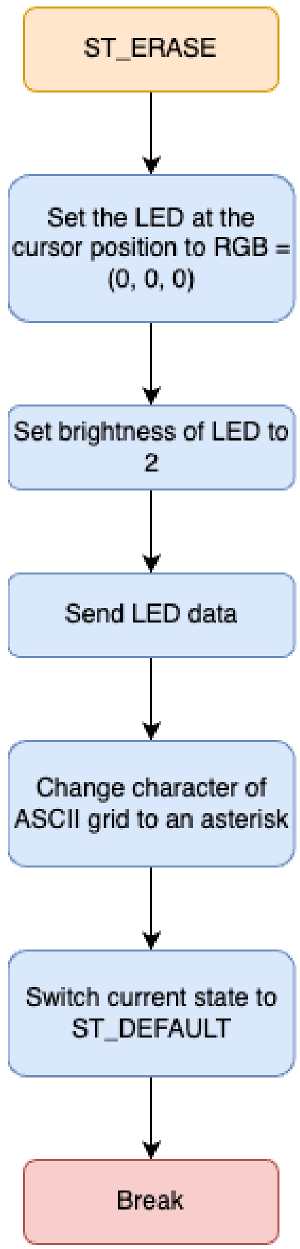

Figure 9. ST_ERASE flowchart

ST_CLEAR

This state is functionally identical to ST_CLEAR, except we use **Reset_LED** to iterate over all the LEDs and clear them. Then, we call **Set_Brightness** and **WS2812_Send**, and reset the entire ASCII matrix to asterisk characters. We change the current state to ST_DEFAULT.

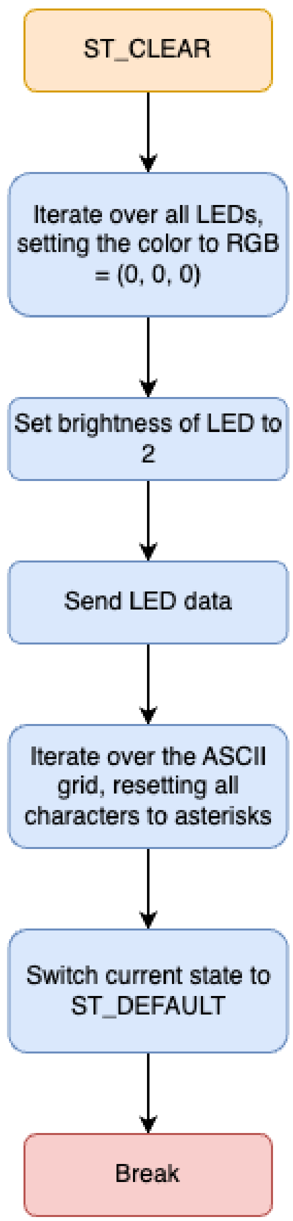

Figure 10. ST_CLEAR flowchart

5. UART

UART_Init

This function configures pins PA2 and PA3 for alternate function mode, as those two pins correspond to the transmit and receive registers for USART2. The function also configures USART2, enabling transmit mode, receive mode, as well as interrupts. We also set USART2 to LSB first with 1 stop bit.

Additionally, we set the baud rate register. For a target baud rate of 115200, the baud rate divisor can be calculated as follows:

$$Baud\ rate\ divisor\ =\ ceil(\frac{Clock\ Speed}{Baud\ rate})\ =\ ceil(\frac{24*10^6}{115200})\ =\ 208$$

Finally, we enable USART2, as well as interrupts in the NVIC.

UART_print

This function takes in an input string (char*) and transmits it via UART. We iterate over the characters of the string and set the USART2 transmit data register (TDR) equal to the current character. Finally, the function waits for the transmission to be complete by checking if the interrupt flag is set high.

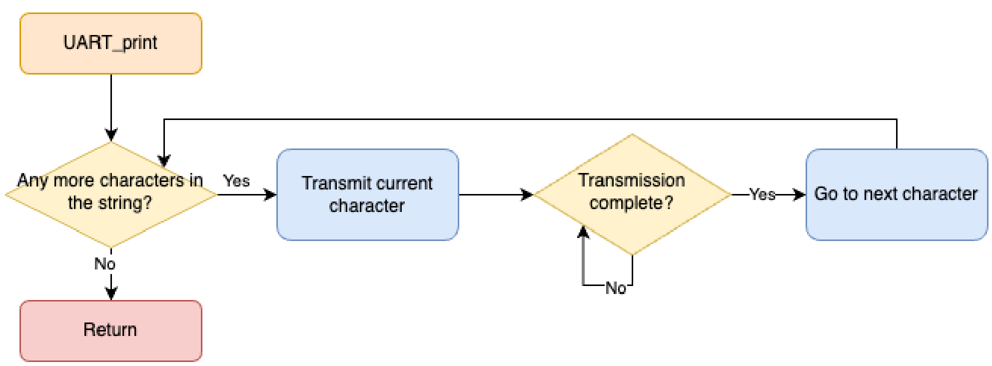

Figure 11. UART_print flowchart

UART_print_char

This function differs slightly from UART_print, in that it takes an integer as an input instead of a string. This allows for easier transmission of singular characters, as character inputs can be converted into integers seamlessly.

The function follows the same process as UART_print, in that it sets USART2->TDR equal to the input character, and waits for the transmission to finish.

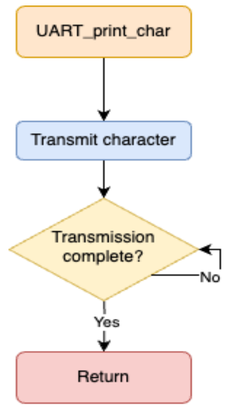

Figure 12. UART_print_char flowchart

UART_ESC_Code

This function is identical to UART_print, except that it first sets USART2->TDR equal to 0x1B, indicating that we are about to transmit an escape code. For our device, these escape codes allow us to easily clear the terminal and reset the cursor.

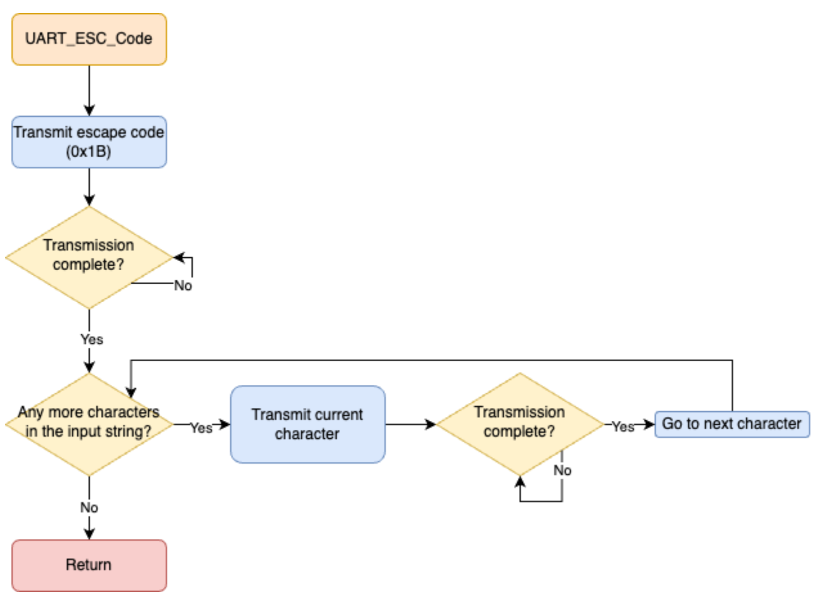

Figure 13. UART_ESC_Code flowchart

UART_print_int

This function takes in an input integer, and prints it to the terminal. This is done by initializing a character array of integers 0 through 9 and calling our UART_print_char function on intArray[input] (this returns the string value of the number).

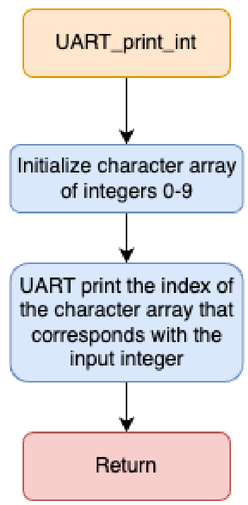

Figure 14. UART_print_int flowchart

USART2_IRQHandler

This function is called every time a USART2 interrupt is triggered. In this function, we check if the USART2 read data register is not empty. If the register isn't empty, we store the contents of the read data register (in our case, the key press) in our global key press variable. Finally, we set our key pressed flag to 1, indicating that a key has been pressed to our main program loop.

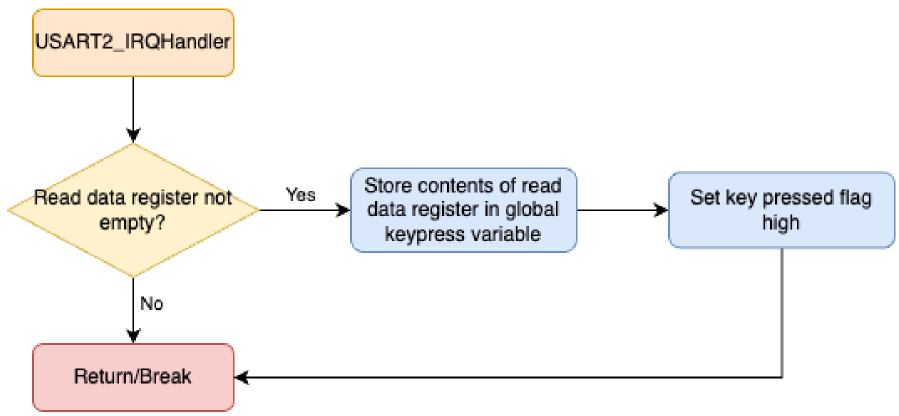

Figure 15. USART2_IRQHandler flowchart

8. Other Functions

print_matrix

This function takes in the x and y coordinate of the terminal cursor as inputs, and prints the ASCII grid with the current cursor position. This is done by first iterating over the global **UI_matrix** and printing its character elements with **UART_print_char**. When printing across the rows, I separate each element with a space character for easier visibility.

Once the matrix is printed, I use **UART_ESC_Code** to move the cursor right ‘x’ times, and down ‘y’ times. When moving the cursor right, we move it right twice per iteration to account for the space characters specified above.

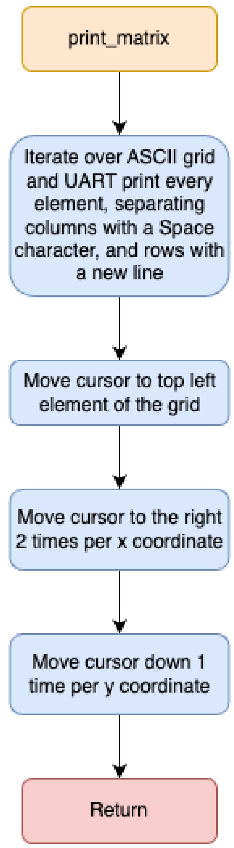

Figure 16. print_matrix flowchart

Power Calculations

This device uses the STM32, the terminal using UART communications, a GPIO pin, and a 5V power supply. Using standard Energizer E95BP-2 batteries, we will exceed our 5 V limit with four batteries, which is why we will need a L7805 DC voltage regulator. The power diagram is as follows:

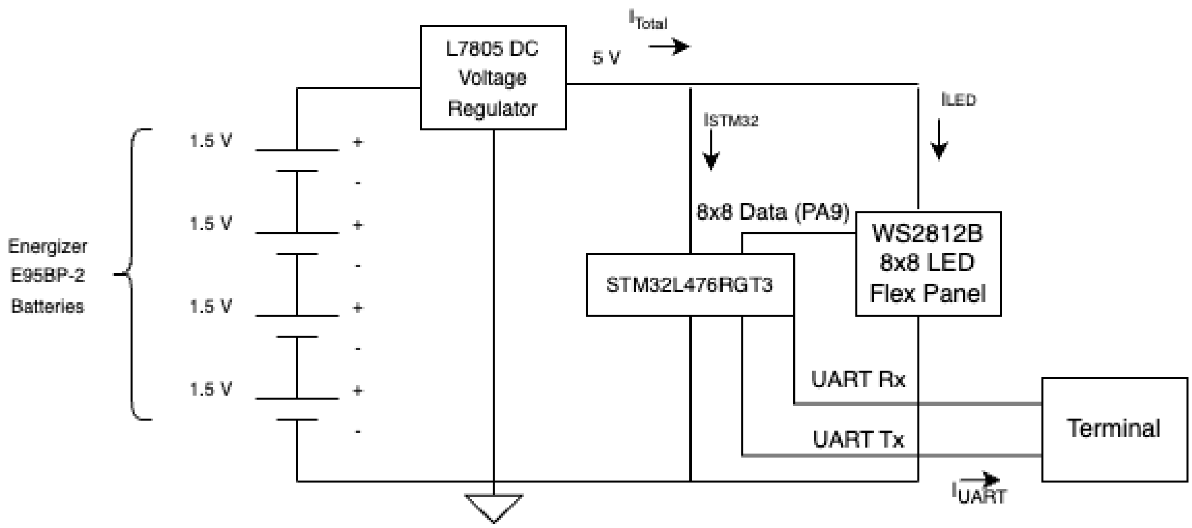

Figure 17. 8x8 Draw Power Block Diagram

Here is a list of the device components, and how much current they use:

Component	Current Usage
STM32L476RGT3 (72 MHz clock)	10.16 µA (25℃)
WS2812B 8x8 LED Flex Panel	50 mA / LED (full brightness)
UART	1.4 µA
GPIO Pin PA9	20 mA (max.)

Table 2. Device component current usage [2][5]

The LED Flex Panel uses the most current when all 64 LEDs are white (RGB = 255, 255, 255). For our calculations, we will assume the user draws an image on the LED Panel using all 64 LEDS, over a span of 5 minutes. Therefore, we require 50 mA * 64 = 3.2 Amps for all 64 LEDs to be powered on. The value of 50 mA is used as a worst-case calculation.

With I_{STM32} = 10.16 μA, I_{LED} = 3.2 A, I_{UART} = 1.4 μA, I_{GPIO} = 20 mA, and T = 300 s (5 minutes), our calculation for I_{Total} is:

$$I_{Total} = T * (I_{STM32} + I_{UART} + I_{LED} + I_{GPIO})$$

$$I_{Total} = 300 * (10.16\,\mu A + 1.4\,\mu A + 3.2\,A + 20\,mA) = 960.703\,A$$

To find the average current consumption, we do:

$$I_{Avg} = \frac{I_{Total}}{T} = \frac{960.703}{300} = 3.229\,A$$

To calculate battery life, we will need the battery current as well as the battery capacity. Assuming an efficiency of 83% for our L7805 (given we are using a lot of current and voltage), we can solve for the battery current by equating the total power of the left side of the voltage regulator to the total power the right side (see **Figure 17**):

$$4 * (1.5\,V) * 0.83 * I_{Battery} = 5\,V * 3.229\,A$$

$$I_{Battery} = \frac{5\,V * 3.229\,A}{4*(1.5\,V)*0.83} = 3.24\,A$$

Finally, we need the battery capacity. Since we are using *a lot* of current, I will use a battery capacity estimate of 9000 mA hours (or 9 Amp hours):

$$T_{Battery} = \frac{Battery\ capacity}{I_{Battery}} = \frac{9\ Amp\ hours}{3.24\,A} = 2.78\,hours$$

Our total battery life is roughly 2 hours, 47 minutes.

Users Manual

Setting up the Device

To use the 8x8 LED Draw application, you will need a computer with a USB port.

1. Connect your computer to the device using the USB to Mini-B cable.
2.
    a. **Windows**
        i. On Windows, you can use the Real Term application to simulate a terminal https://realterm.i2cchip.com/.
        ii. Configure the terminal to display as Ansi-VT100 and set the baud rate to 115200 kbps. The application should start. Note that you may need to press the black "reset" knob on the device (STM32).

    b. **OSX**
        i. On Mac, you can use the built in terminal to launch 8x8 LED Draw.
        ii. Launch the terminal app, then type **ls /dev/cu.*** and press Enter.
        iii. Locate the connected device in the list. This may require unplugging the device and re-typing the command to see which device disappears from the list. On my computer, the device appeared as **/dev/cu.usbmodemXXXX**.
        iv. Type **screen /dev/cu.name_of_device 115200** and press Enter. The application should start. Note that you may need to press the black "reset" knob on the device (STM32).

Controls

Home Screen

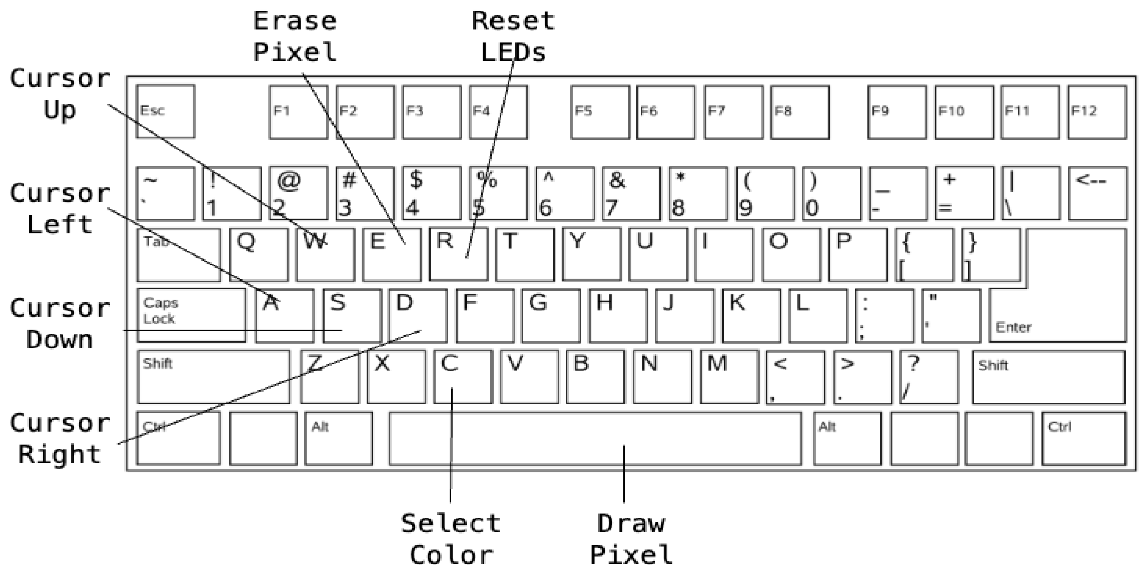

Figure 17. Keyboard Controls for 8x8 LED Draw, *Home Screen*

Select Color Screen

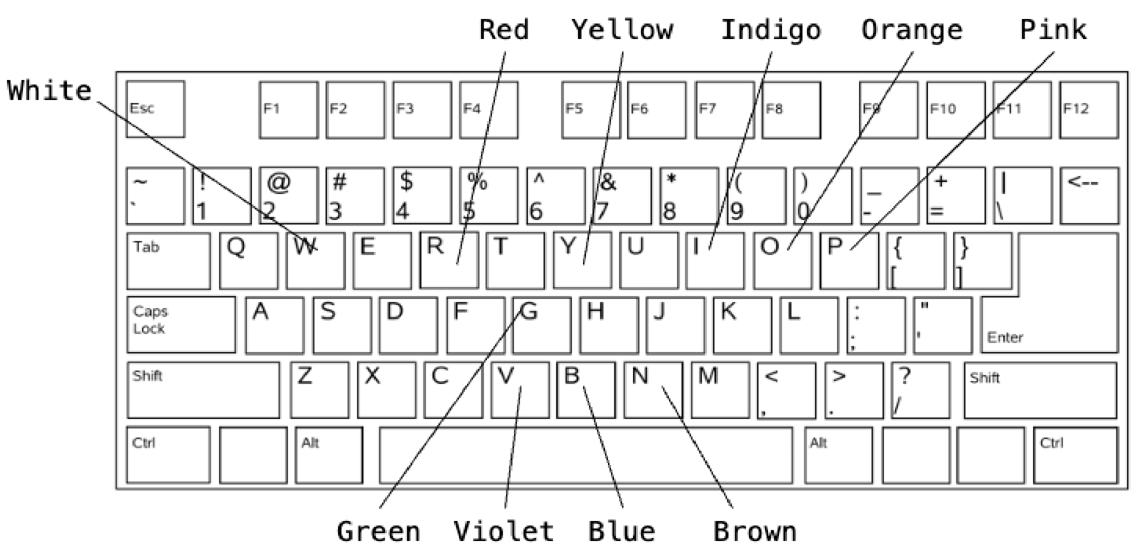

Figure 18. Keyboard Controls for 8x8 LED Draw, *Select Color Screen*

Using the Application

Upon startup, the user is presented with the home screen interface, showing the controls of the program, as well as an 8x8 grid consisting of asterisks (an asterisk represents no color). This grid allows the user to see exactly which pixel (LED) they are drawing on. Below the controls section, the interface displays the current color the user is drawing with, as well as the current x and y coordinate of the pixel being drawn on (coordinate [1,1] is the top left of the LED board, and [8,8] is the bottom right of the board). The home screen interface is shown below:

```
Welcome to 8x8 Draw!

Controls:
W         Cursor Up
A         Cursor Left
S         Cursor Down
D         Cursor Right
C         Select Color
Space     Draw Pixel
E         Erase Pixel
R         Reset LEDs

Color: WHITE
x: 1, y: 1

* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

Figure 19. Home Screen

Upon pressing W, A, S, or D, the terminal cursor will move up, left, down, or right on the asterisk grid.

Upon pressing C, the application will switch to a different interface, allowing the user to select the drawing color:

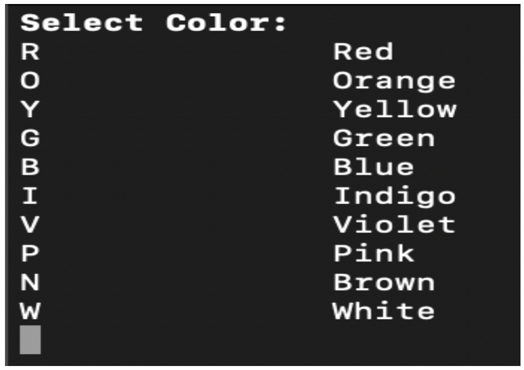

Figure 20. Select Color Screen

Once a color is chosen, the application returns to the home screen, reflecting the color the user selected.

Upon pressing Spacebar, the pixel that the cursor is hovering over is replaced by the keyboard character of the current drawing color. Additionally, the corresponding LED on the LED matrix will turn on and present the respective color:

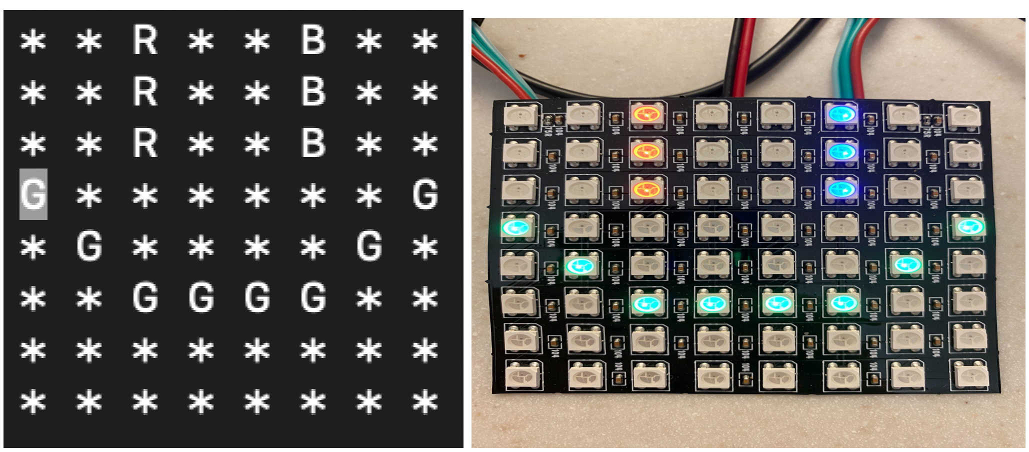

Figure 21. Drawing on ASCII grid (left) and LED board (right)

Upon pressing E, the pixel that the cursor is hovering over is replaced by an asterisk, and the corresponding LED will be turned off, representing an eraser.

Upon pressing R, all pixels on the ASCII grid are replaced by asterisks, and all LEDs are turned off, representing a "clear" function.

Documents Referenced

[1] P. Hummel and J. Gerfen, “STM32 Lab Manual,” Google Docs, Accessed: June 7, 2023 [Online]. Available: https://docs.google.com/document/d/1Btl--IQGtYRRn8naFpLwn64Av9y7no5OkXmoK1pFN4g/edit#heading=h.z6v22gj9iqm0

[2] “ST32L476xx Data Sheet,” ST.com., Accessed: Apr. 30, 2023 [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l476rg.pdf

[3] “ST32L476 Reference Manual,” ST.com., Accessed: June 7, 2023 [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxxstm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcusstmicroelectronics.pdf.

[4] “Interface WS2812 with STM32,” ControllersTech.com., Accessed: June 7, 2023 [Online]. Available: https://controllerstech.com/interface-ws2812-with-stm32/

[5] “WS2812B Datasheet,” SGBiotic.com., Accessed: June 7, 2023 [Online]. Available: https://www.sgbotic.com/products/datasheets/display/WS2812B.pdf

```c
#include "main.h"
#include "math.h"
#include "uart.h"
#include "LED_matrix.h"

#define DEFAULT_BRIGHTNESS 1// The brightness value used for this program, anything higher is too bright

uint8_t LED_Data[MAX_LED][4];   // Stores LED number and RGB
uint8_t LED_Mod[MAX_LED][4];// Stores LED number and RGB, used only when USE_BRIGHTNESS is enabled (1)

// Initialize user interface grid
char UI_Matrix[MATRIX_WIDTH][MATRIX_HEIGHT] = {
        { '*', '*', '*', '*', '*', '*', '*', '*' },
        { '*', '*', '*', '*', '*', '*', '*', '*' },
        { '*', '*', '*', '*', '*', '*', '*', '*' },
        { '*', '*', '*', '*', '*', '*', '*', '*' },
        { '*', '*', '*', '*', '*', '*', '*', '*' },
        { '*', '*', '*', '*', '*', '*', '*', '*' },
        { '*', '*', '*', '*', '*', '*', '*', '*' },
        { '*', '*', '*', '*', '*', '*', '*', '*' }
};

// Initialize LED numbers to easily reference with x and y coordinates
uint8_t LED_Matrix[MATRIX_WIDTH][MATRIX_HEIGHT] = {
        { 0, 15, 16, 31, 32, 47, 48, 63 },
        { 1, 14, 17, 30, 33, 46, 49, 62 },
        { 2, 13, 18, 29, 34, 45, 50, 61 },
        { 3, 12, 19, 28, 35, 44, 51, 60 },
        { 4, 11, 20, 27, 36, 43, 52, 59 },
        { 5, 10, 21, 26, 37, 42, 53, 58 },
        { 6, 9,  22, 25, 38, 41, 54, 57 },
        { 7, 8,  23, 24, 39, 40, 55, 56 }
};

int datasentflag = 0;       // indicates when data has been sent to LEDs

TIM_HandleTypeDef htim1;
DMA_HandleTypeDef hdma_tim1_ch2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_TIM1_Init(void);

void print_matrix(uint8_t x_pos, uint8_t y_pos);

/*
 * Function called when DMA pulse is completed
 *
 * Source:
 * https://controllerstech.com/interface-ws2812-with-stm32/
 */
void HAL_TIM_PWM_PulseFinishedCallback(TIM_HandleTypeDef *htim) {
    HAL_TIM_PWM_Stop_DMA(&htim1, TIM_CHANNEL_2);                // Stop DMA transfer
    datasentflag = 1;                                               // Set data flag high
}

// Pulse width modulation data, stores 60 or 30, based on if data is a 1 or 0
uint16_t pwmData[(24 * MAX_LED) + 50];

/*
 * Function to send data to LEDs
 *
 * Source:
 * https://controllerstech.com/interface-ws2812-with-stm32/
 */
void WS2812_Send(void) {
    uint32_t indx = 0;
    uint32_t color;

    // Concatenate the color bits
    for (int i = 0; i < MAX_LED; i++) {
#if USE_BRIGHTNESS
        color = ((LED_Mod[i][1] << 16) | (LED_Mod[i][2] << 8) | (LED_Mod[i][3]));
#else
        color = ((LED_Data[i][1]<<16) | (LED_Data[i][2]<<8) | (LED_Data[i][3]));
#endif
        // write 60 if color bit is a 1, otherwise write 30
        for (int i = 23; i >= 0; i--) {
            if (color & (1 << i)) {
                pwmData[indx] = 60;  // 2/3 (~68%) of 90
            }

            else
                pwmData[indx] = 30;  // 1/3 (~32%) of 90

            indx++;
        }

    }

    // 50 us "reset code" after data has been sent
    for (int i = 0; i < 50; i++) {
        pwmData[indx] = 0;
        indx++;
    }
```

```

    HAL_TIM_PWM_Start_DMA(&htim1, TIM_CHANNEL_2, (uint32_t*) pwmData, indx);// Send data
    while (!datasentflag);                                                      // wait for DMA to finish
    datasentflag = 0;                                                           // reset flag
}

int8_t x = 0, y = 0;    // current x and y coordinates on LED board
int key_pressed_flag = 0;   // indicates whether a key has been pressed
char keypress;              // stores value of keypress

/**
 * @brief  The application entry point.
 * @retval int
 */
int main(void) {

    HAL_Init();

    SystemClock_Config();

    // Init TIM1 Channel 2 for PWM and DMA (PA9)
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_TIM1_Init();

    // Init UART
    UART_Init();

    // Start with all LEDs turned off
    Reset_LED();
    Set_Brightness(DEFAULT_BRIGHTNESS);
    WS2812_Send();

    // initialize FSM states
    typedef enum {
        ST_DEFAULT, ST_MOVE_CURSOR, ST_SEL_COLOR, ST_DRAW, ST_ERASE, ST_CLEAR
    } state_var_type;

    // struct variable to store color info
    typedef struct {
        char name[10];  // color name
        uint8_t red;// color R value, 0-255
        uint8_t green;  // color G value, 0-255
        uint8_t blue;   // color B value, 0-255
        char key;       // keypress associated with color switch
    } color;

    // Initialize usable colors
    color RED = { "RED", 255, 0, 0, 'R' };
    color ORANGE = { "ORANGE", 255, 127, 0, 'O' };
    color YELLOW = { "YELLOW", 255, 255, 0, 'Y' };
    color GREEN = { "GREEN", 0, 255, 0, 'G' };
    color BLUE = { "BLUE", 0, 0, 255, 'B' };
    color INDIGO = { "INDIGO", 75, 0, 130, 'I' };
    color VIOLET = { "VIOLET", 148, 0, 211, 'V' };
    color PINK = { "PINK", 255, 105, 180, 'P' };
    color BROWN = { "BROWN", 150, 75, 0, 'N' };
    color WHITE = { "WHITE", 255, 255, 255, 'W' };

    // Initial state and color
    state_var_type curr_state = ST_DEFAULT;
    color curr_color = WHITE;

    while (1) {
        switch (curr_state) {

        // Default state which prints User Interface
        case ST_DEFAULT:
            UART_ESC_Code("[2J");                           // clear terminal
            UART_ESC_Code("[H");                        // move cursor to top left
            UART_ESC_Code("[1m");                           // bold text
            UART_ESC_Code("[34m");                          // blue text
            UART_print("Welcome to 8x8 Draw!\r\n\r\n");     // print title
            UART_ESC_Code("[0m");                           // turn off character attributes

            // Print controls section
            UART_print("Controls:\r\n");
            UART_print("W         Cursor Up\r\n");
            UART_print("A         Cursor Left\r\n");
            UART_print("S         Cursor Down\r\n");
            UART_print("D         Cursor Right\r\n");
            UART_print("C         Select Color\r\n");
            UART_print("Space     Draw Pixel\r\n");
            UART_print("E         Erase Pixel\r\n");
            UART_print("R         Reset LEDs\r\n");
            UART_print("\r\n");
            UART_print("Color: ");
            UART_print(curr_color.name);
            UART_print("\r\n");
            UART_print("x: ");
            UART_print_int(x + 1);
            UART_print(", y: ");
            UART_print_int(y + 1);

            // print UI grid, using current x and y values for the cursor placement
            UART_print("\r\n\r\n");
```

```c
            print_matrix(x, y);

            while (!key_pressed_flag);      // wait for key to be pressed
            key_pressed_flag = 0;           // clear key press flag

            // switch FSM state based on keypress
            if (keypress == 'w' || keypress == 'a' || keypress == 's' || keypress == 'd') {     // if W, A, S, D, move cursor
                curr_state = ST_MOVE_CURSOR;
            } else if (keypress == 'c') {                                                       // if C, select color
                curr_state = ST_SEL_COLOR;
            } else if (keypress == ' ') {                                                       // if Space, draw pixel
                curr_state = ST_DRAW;
            } else if (keypress == 'e') {                                                       // if E, erase pixel
                curr_state = ST_ERASE;
            } else if (keypress == 'r') {                                                       // if R, reset LEDs
                curr_state = ST_CLEAR;
            } else {                                                                        // otherwise, don't switch state
                curr_state = ST_DEFAULT;
            }
            break;

        // State to move cursor based on keypress
        case ST_MOVE_CURSOR:

            // Increment of decrement x/y coordinate, depending on the key pressed
            // Make sure 0 <= x,y <= 7
            switch (keypress) {
            case 'w':
                y--;
                y = (y < 0) ? 0 : y;
                break;
            case 'a':
                x--;
                x = (x < 0) ? 0 : x;
                break;
            case 's':
                y++;
                y = (y > 7) ? 7 : y;
                break;
            case 'd':
                x++;
                x = (x > 7) ? 7 : x;
                break;
            default:
                break;
            }
            curr_state = ST_DEFAULT;// go to default state, re-print UI
            break;

        // State to select color
        case ST_SEL_COLOR:
            // Print select color interface
            UART_ESC_Code("[2J");                       // clear terminal
            UART_ESC_Code("[H");                    // move cursor to top left
            UART_ESC_Code("[5m");                       // blinking text
            UART_ESC_Code("[1m");                       // bold text
            UART_print("Select Color: \r\n");
            UART_ESC_Code("[0m");

            // Print keypress with respective color
            UART_print("R              Red\r\n");
            UART_print("O              Orange\r\n");
            UART_print("Y              Yellow\r\n");
            UART_print("G              Green\r\n");
            UART_print("B              Blue\r\n");
            UART_print("I              Indigo\r\n");
            UART_print("V              Violet\r\n");
            UART_print("P              Pink\r\n");
            UART_print("N              Brown\r\n");
            UART_print("W              White\r\n");

            while (!key_pressed_flag);              // wait for key to be pressed
            key_pressed_flag = 0;                   // clear key press flag

            // Change color based on the key pressed
            switch (keypress) {
            case 'r':
                curr_color = RED;
                break;
            case 'o':
                curr_color = ORANGE;
                break;
            case 'y':
                curr_color = YELLOW;
                break;
            case 'g':
                curr_color = GREEN;
                break;
            case 'b':
                curr_color = BLUE;
                break;
            case 'i':
                curr_color = INDIGO;
                break;
            case 'v':
                curr_color = VIOLET;
```

```c
                break;
            case 'p':
                curr_color = PINK;
                break;
            case 'n':
                curr_color = BROWN;
                break;
            case 'w':
                curr_color = WHITE;
                break;
            default:
                break;
            }
            curr_state = ST_DEFAULT;// go to default state, re-print UI
            break;

        // State to draw pixel
        case ST_DRAW:
            Set_LED(LED_Matrix[y][x], curr_color.red, curr_color.green, curr_color.blue);   // Set LED using x, y, and current color
            Set_Brightness(DEFAULT_BRIGHTNESS);                                                         // Set brightness of LED
            WS2812_Send();                                                                  // Send LED data
            UI_Matrix[x][y] = curr_color.key;                                               // Change char on UI Grid to color keypress
            curr_state = ST_DEFAULT;                                                    // go to default state, re-print UI
            break;

        // State to erase pixel
        case ST_ERASE:
            Set_LED(LED_Matrix[y][x], 0, 0, 0);     // Clear LED with (R, G, B) = (0, 0, 0)
            Set_Brightness(DEFAULT_BRIGHTNESS);     // Set brightness of LED (must be done for DMA to work)
            WS2812_Send();                          // Send LED data
            UI_Matrix[x][y] = '*';                  // Change char on UI Grid to '*'
            curr_state = ST_DEFAULT;            // go to default state, re-print UI
            break;

        // State to clear all LEDs
        case ST_CLEAR:
            Reset_LED();                        // Reset LEDs
            Set_Brightness(DEFAULT_BRIGHTNESS);     // Set brightness (must be done for DMA to work)
            WS2812_Send();                          // Send LED Data

            // Re-init UI Grid to '*' characters
            for (int i = 0; i < MATRIX_WIDTH; i++) {
                for (int j = 0; j < MATRIX_HEIGHT; j++) {
                    UI_Matrix[i][j] = '*';
                }
            }

            curr_state = ST_DEFAULT;            // go to default state, re-print UI
            break;
        }
    }
    /* USER CODE END 3 */
}

/* prints the UI Grid and adjusts terminal cursor based on x and y coordinates */
void print_matrix(uint8_t x_pos, uint8_t y_pos) {
    // Print asterisks
    for (int i = 0; i < MATRIX_WIDTH; i++) {
        for (int j = 0; j < MATRIX_HEIGHT; j++) {
            UART_print_char(UI_Matrix[j][i]);
            UART_print_char(' ');
        }
        UART_print("\r\n");
    }

    UART_ESC_Code("[8A");                       // Start cursor at top left of the grid

    // Move cursor x times to the right
    for (int m = 0; m < x_pos; m++) {
        UART_ESC_Code("[2C");                   // Move cursor 2 positions right (to account for spaces between asterisks)
    }

    // Move cursor y times down
    for (int n = 0; n < y_pos; n++) {
        UART_ESC_Code("[1B");                   // Move cursor 1 position down
    }
}

/* Interrupt handler for USART2 */
void USART2_IRQHandler(void) {
    // If read data register not empty, store the data (keypress) in global variable and set key press flag high
    if ((USART2->ISR & USART_ISR_RXNE) != 0) {
        keypress = USART2->RDR;
        key_pressed_flag = 1;
    }

}

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void) {
    RCC_OscInitTypeDef RCC_OscInitStruct = { 0 };
    RCC_ClkInitTypeDef RCC_ClkInitStruct = { 0 };

```

```c
    /** 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_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
    RCC_OscInitStruct.PLL.PLLM = 1;
    RCC_OscInitStruct.PLL.PLLN = 9;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
    RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
    RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
    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_PLLCLK;
    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_4) != HAL_OK) {
        Error_Handler();
    }
}

/**
 * @brief TIM1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM1_Init(void) {

    /* USER CODE BEGIN TIM1_Init 0 */

    /* USER CODE END TIM1_Init 0 */

    TIM_MasterConfigTypeDef sMasterConfig = { 0 };
    TIM_OC_InitTypeDef sConfigOC = { 0 };
    TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = { 0 };

    /* USER CODE BEGIN TIM1_Init 1 */

    /* USER CODE END TIM1_Init 1 */
    htim1.Instance = TIM1;
    htim1.Init.Prescaler = 0;
    htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim1.Init.Period = 90 - 1;
    htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim1.Init.RepetitionCounter = 0;
    htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_PWM_Init(&htim1) != HAL_OK) {
        Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig)
            != HAL_OK) {
        Error_Handler();
    }
    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = 0;
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
    sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2)
            != HAL_OK) {
        Error_Handler();
    }
    sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;
    sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
    sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
    sBreakDeadTimeConfig.DeadTime = 0;
    sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
    sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
    sBreakDeadTimeConfig.BreakFilter = 0;
    sBreakDeadTimeConfig.Break2State = TIM_BREAK2_DISABLE;
    sBreakDeadTimeConfig.Break2Polarity = TIM_BREAK2POLARITY_HIGH;
    sBreakDeadTimeConfig.Break2Filter = 0;
    sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
    if (HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig)
```

```c
            != HAL_OK) {
        Error_Handler();
    }
    /* USER CODE BEGIN TIM1_Init 2 */

    /* USER CODE END TIM1_Init 2 */
    HAL_TIM_MspPostInit(&htim1);

}

/**
 * Enable DMA controller clock
 */
static void MX_DMA_Init(void) {

    /* DMA controller clock enable */
    __HAL_RCC_DMA1_CLK_ENABLE();

    /* DMA interrupt init */
    /* DMA1_Channel3_IRQn interrupt configuration */
    HAL_NVIC_SetPriority(DMA1_Channel3_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(DMA1_Channel3_IRQn);

}

/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void) {
    /* USER CODE BEGIN MX_GPIO_Init_1 */
    /* USER CODE END MX_GPIO_Init_1 */

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOA_CLK_ENABLE();

    /* USER CODE BEGIN MX_GPIO_Init_2 */
    /* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
 * @brief  This function is executed in case of error occurrence.
 * @retval None
 */
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
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
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
/*
 * LED_matrix.h
 *
 *  Created on: Jun 7, 2023
 *      Author: noahmasten
 */

#ifndef SRC_LED_MATRIX_H_
#define SRC_LED_MATRIX_H_

#include <stdint.h>

#define MAX_LED 64          // number of LEDs
#define USE_BRIGHTNESS 1// 1 or 0, depending on if we want to change brightness or not
#define MATRIX_WIDTH 8      // width of LED panel
#define MATRIX_HEIGHT 8     // height of LED panel
#define PI 3.14159265

extern uint8_t LED_Matrix[MATRIX_WIDTH][MATRIX_HEIGHT];
extern uint8_t LED_Data[MAX_LED][4];
extern uint8_t LED_Mod[MAX_LED][4];

void Set_LED (int LEDnum, int Red, int Green, int Blue);
void Set_Brightness (int brightness);
void Reset_LED (void);

#endif /* SRC_LED_MATRIX_H_ */
```

```c
/*
 * LED_matrix.c
 *
 *  Created on: Jun 7, 2023
 *      Author: noahmasten
 */

#include "LED_Matrix.h"
#include "math.h"
#include <stdint.h>

/*
 * Stores LED number and RGB values in array
 *
 * Source:
 * https://controllerstech.com/interface-ws2812-with-stm32/
 */
void Set_LED (int LEDnum, int Red, int Green, int Blue)
{
    LED_Data[LEDnum][0] = LEDnum;
    LED_Data[LEDnum][1] = Green;
    LED_Data[LEDnum][2] = Red;
    LED_Data[LEDnum][3] = Blue;
}

/*
 * Sets brightness of LED
 *
 * Source:
 * https://controllerstech.com/interface-ws2812-with-stm32/
 */
void Set_Brightness (int brightness)  // 0-45
{
#if USE_BRIGHTNESS

    if (brightness > 45) brightness = 45;
    for (int i=0; i<MAX_LED; i++)
    {
        LED_Mod[i][0] = LED_Data[i][0];
        for (int j=1; j<4; j++)
        {
            float angle = 90-brightness;  // in degrees
            angle = angle*PI / 180;  // in rad
            LED_Mod[i][j] = (LED_Data[i][j])/(tan(angle));
        }
    }

#endif
}

/*
 * Resets all LEDs to RGB value (0, 0, 0)
 *
 * Source:
 * https://controllerstech.com/interface-ws2812-with-stm32/
 */
void Reset_LED (void)
{
    for (int i=0; i<MAX_LED; i++)
    {
        LED_Data[i][0] = i;
        LED_Data[i][1] = 0;
        LED_Data[i][2] = 0;
        LED_Data[i][3] = 0;
    }
}
```

```
/*
 * uart.h
 *
 *  Created on: May 2, 2023
 *      Author: noahmasten
 */

#ifndef SRC_UART_H_
#include "stm32l476xx.h"

#define SRC_UART_H_

#define BAUD_RATE 115200 // 115.2 kpbs
#define USART_DIV 625 // clock frequency divided by baud rate, rounded up

void UART_Init(void);
void UART_print(char* data);
void UART_ESC_Code(char *input_string);
void UART_print_char(int input_char);
void UART_print_int(uint8_t integer);

#endif /* SRC_UART_H_ */

```

```c
/*
 * uart.c
 *
 *  Created on: May 3, 2023
 *      Author: noahmasten
 */
#include "uart.h"
#include <stdio.h>
#include <string.h>

/* CONFIGURES PINS PA2 (TX) and PA3 (RX) for USART */
void UART_Init(void) {

    /* enable clock for GPIOA and USART2 */
        RCC->AHB2ENR        |=  (RCC_AHB2ENR_GPIOAEN);
        RCC->APB1ENR1       |= (RCC_APB1ENR1_USART2EN);

        /* GPIO config (PA2->Tx and PA3->Rx) */
        GPIOA->AFR[0]   &= ~(GPIO_AFRL_AFSEL2_Msk | GPIO_AFRL_AFSEL3_Msk);  // clear AFR
        GPIOA->AFR[0]       |=  ( (0x7UL << GPIO_AFRL_AFSEL2_Pos) |             // set PA2, PA3 to AF7
                                  (0x7UL << GPIO_AFRL_AFSEL3_Pos) );
        GPIOA->MODER        &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3);          // clear mode2 and mode3
        GPIOA->MODER        |=  (GPIO_MODER_MODE2_1 | GPIO_MODER_MODE3_1);      // set to alternate function
        GPIOA->OTYPER       &= ~(GPIO_OTYPER_OT2 | GPIO_OTYPER_OT3);        // set OTYPE2 and OTYPE3 to push-pull
        GPIOA->OSPEEDR      |=  (GPIO_OSPEEDR_OSPEED2 | GPIO_OSPEEDR_OSPEED3);  // set OSPEED2 and OPSEED3 to high speed
        GPIOA->PUPDR        &= ~(GPIO_PUPDR_PUPD2 | GPIO_PUPDR_PUPD3);          // set PUPD2 and PUPD3 to no pull-up/pull-down

        /* USART2 config */
        USART2->CR1 &= ~(USART_CR1_M);                                      // set word length to 8 bits
        USART2->CR1 |=  (USART_CR1_TE | USART_CR1_RE | USART_CR1_RXNEIE);   // transmit enable, read enable, RXNE interrupt enable
        USART2->CR2 &= ~(USART_CR2_MSBFIRST | USART_CR2_STOP);              // LSB first, 1 stop bit
        USART2->BRR  =  (USART_DIV);                                    // baud rate configuration
        USART2->CR1 |=  (USART_CR1_UE);                                     // enable USART2

        /* enable interrupts in NVIC */
        NVIC->ISER[1] = (1 << (USART2_IRQn & 0x1F));

        /* enable interrupts globally */
        __enable_irq();

}

void UART_print(char *input_string) {
    for(int i = 0; i < strlen(input_string); i++) {
        USART2->TDR = input_string[i];          // transmit character
        while(!(USART2->ISR & USART_ISR_TC));   // wait until transmission is complete
    }
}

/**
  * @brief Transmits escape + input_string
  * @retval None
  */
void UART_ESC_Code(char *input_string) {
    USART2->TDR = 0x1B;                     // transmit escape
    while(!(USART2->ISR & USART_ISR_TC));   // wait until transmission is complete

    for(int i = 0; i < strlen(input_string); i++) {
        USART2->TDR = input_string[i];          // transmit character
        while(!(USART2->ISR & USART_ISR_TC));   // wait until transmission is complete
    }
}

/* Prints single character */
void UART_print_char(int input_char) {
    USART2->TDR = input_char; // transmit character
    while(!(USART2->ISR & USART_ISR_TC)); // wait until transmission is complete
}

/* Prints single integer */
void UART_print_int(uint8_t integer) {
    static char intArray[10] = "0123456789";// char array for easy indexing

    UART_print_char(intArray[integer]);         // print integer
}
```