nucleotype

A Terminal-Based Typing Test

Charles Snead
Brian Nguyen
June 9th, 2023
CPE 316, Section 01
Professor Paul Hummel

Behavior Description

The designed device is a typing speed test that that measures a user's typing speed and accuracy. The device allows you to select between three different timing selections: 10, 15, and 30 seconds. This sets the amount of type you can spend typing before time is up In each case, a series of 50 words is generated to be typed by the user within the selected time period. The timer starts as soon as the user starts typing the first word and an LCD display updates every second to let the user know how much time they have left to type. When time is up, a buzzer goes off to notify the user that they can no longer type. The user then loses the ability to type into the terminal. Finally, results in the form of typing speed in words per minute and accuracy of typed words are displayed for the user. The inspiration for this project comes from the popular typing test website Monkeytype, which also lets you select time modes and tells you your typing speed and accuracy at the end of each test.

System Specification

STM32 NUCLEO-L476RG Specifications	
Supply Voltage	3.3 V
Maximum Current Draw	3.26 mA
Maximum Power Consumption	10.78 mW
System Clock Speed	24 MHz
Terminal and Program Specifications and Commands	
Baud Rate	115200 kbps
Speed Measurement	Words per second (WPS)
Max Measureable Speed	300 WPS (50 words in 10 seconds * 6)
Accuracy Measurement	% (correct words / total words)
Word Bank Size	100 words
“1”	Select “10 seconds mode”
“2”	Select “15 seconds mode”
“3”	Select “30 seconds mode”
LCD Specifications	
Number of Rows	2 rows
Characters Per Row	16 characters
Microcontroller Interface Mode	4-bit
Buzzer Specifications	
Resonant Frequency	2300 Hz (± 300 Hz)
Min. Sound Output at 10 cm	85 dB

Table 1. System Specification

System Schematic

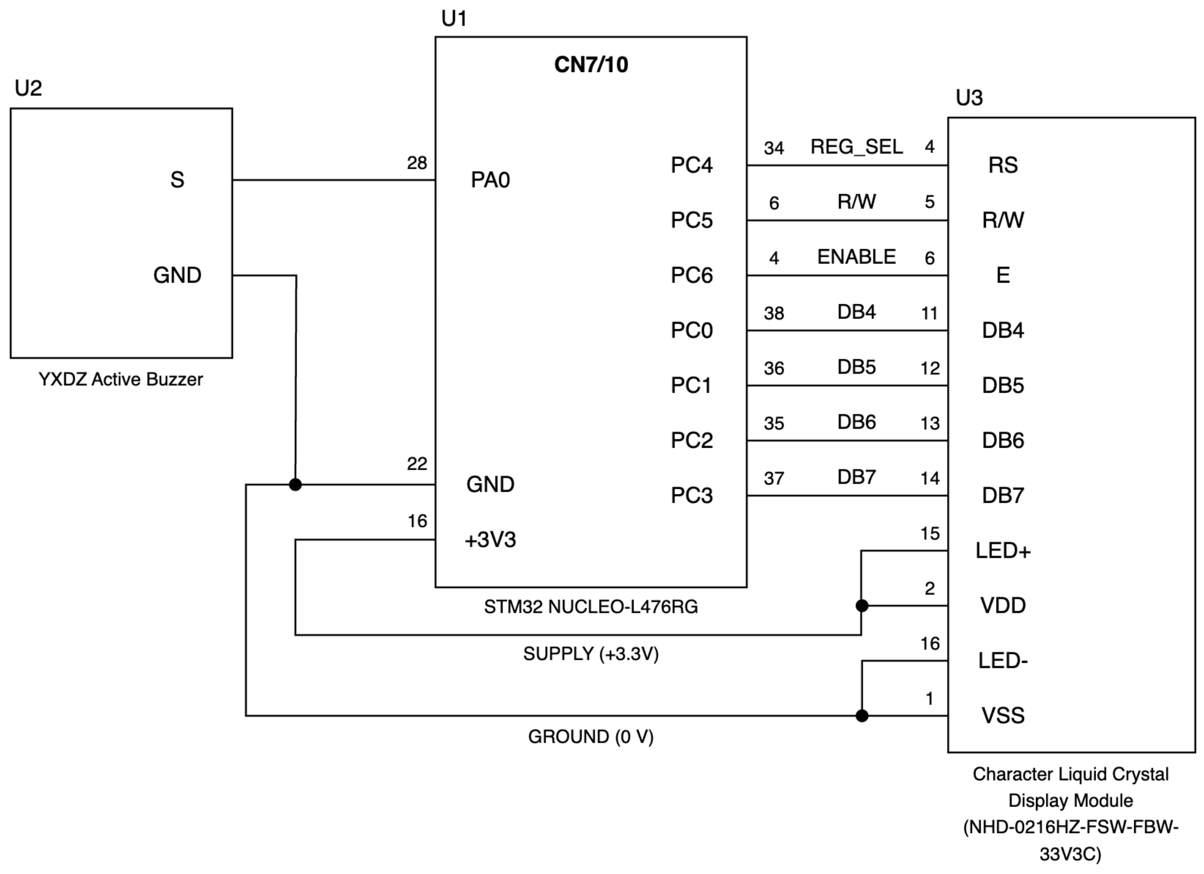


Figure 1. Hardware Connection System Schematic

Software Architecture

The Nucleotype project is designed to interface with UART and other external peripherals to measure the user's typing speed and accuracy in an entertaining matter. Information detailing hardware configuration and software implementation can be found below. Please note that all printing and formatting applications performed in the program use the `UART_print_str()` function and `UART_ESC_code()` function.

Finite State Machine

The Nucleotype functionality is organized into three sections using an FSM, which is implemented in C using switch-case statements. The three states in the FSM are WELCOME_STATE, INPUT_STATE, and RESULT_STATE as seen in Figure 2. The FSM follows a linear path where changing between states occurs once the user presses "ENTER" to switch to the next state.

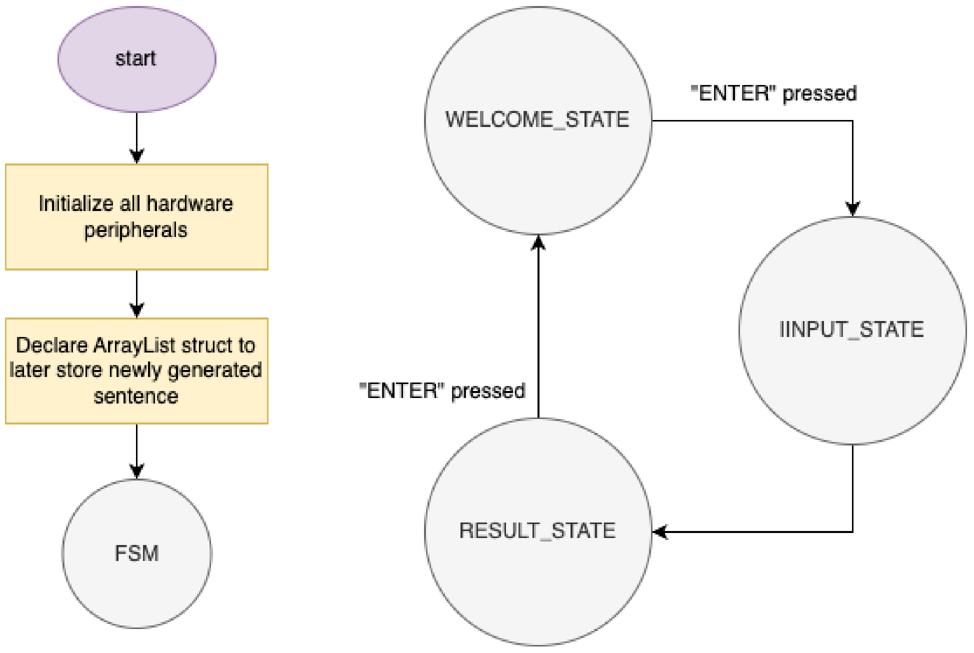
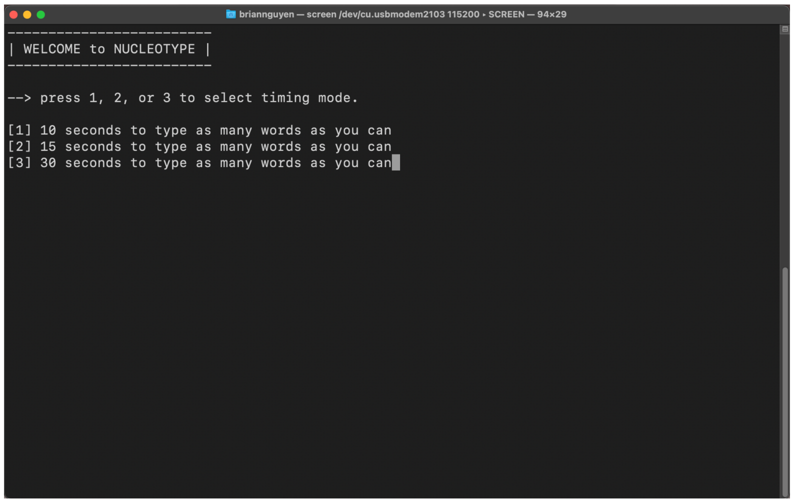

start
Initialize all hardware peripherals
Declare ArrayList struct to later store newly generated sentence
FSM
WELCOME_STATE
"ENTER" pressed
IINPUT_STATE
"ENTER" pressed
RESULT_STATE

Figure 2. Logic for Main and the FSM

Welcome State

On startup, the FSM defaults to the WELCOME_STATE. In this state, the user is greeted with a welcome message and prompted to select the time duration for the typing test by pressing "1" for 10 seconds, "2" for 15 seconds, or "3" for 30 seconds. Once a time duration has been selected, all global variables and timer register values are initialized to the corresponding time in clock cycles. The program then waits for the user to press "ENTER" to switch to the INPUT_STATE.



briannguyen — screen /dev/cu.usbmodem2103 115200 ▸ SCREEN — 94×29

```
-------------------------
| WELCOME to NUCLEOTYPE |
-------------------------

--> press 1, 2, or 3 to select timing mode.

[1] 10 seconds to type as many words as you can
[2] 15 seconds to type as many words as you can
[3] 30 seconds to type as many words as you can
```

Figure 3. Terminal screen capture of WELCOME_STATE

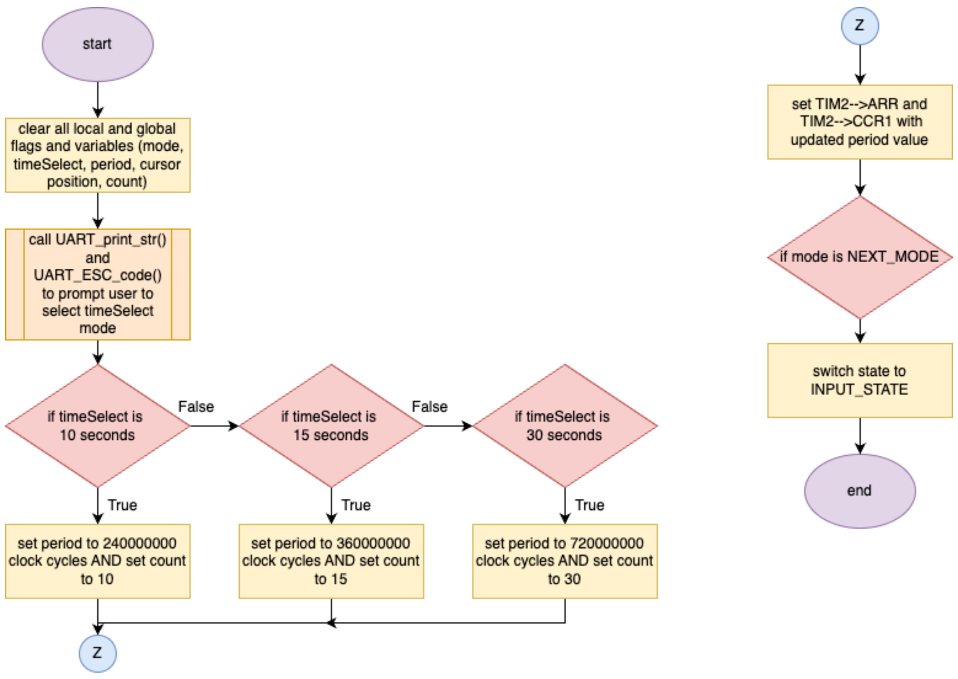
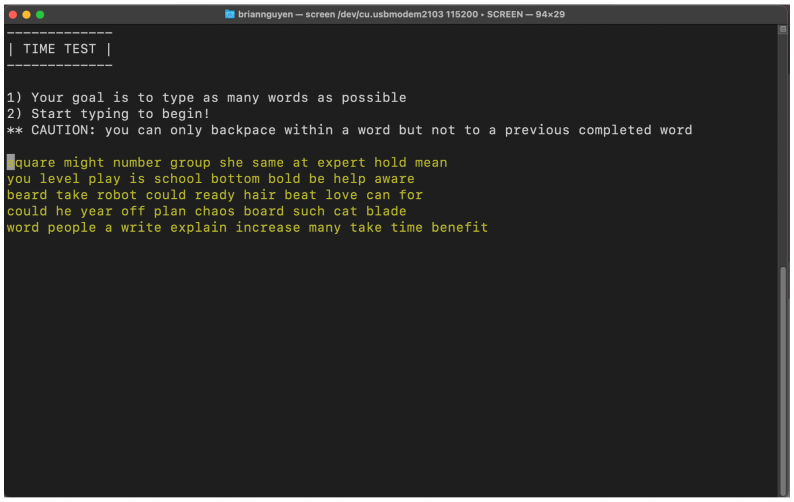

start
clear all local and global flags and variables (mode, timeSelect, period, cursor position, count)
call UART_print_str() and UART_ESC_code() to prompt user to select timeSelect mode
if timeSelect is 10 seconds
False
if timeSelect is 15 seconds
False
if timeSelect is 30 seconds
True
True
True
set period to 240000000 clock cycles AND set count to 10
set period to 360000000 clock cycles AND set count to 15
set period to 720000000 clock cycles AND set count to 30
Z
Z
set TIM2-->ARR and TIM2-->CCR1 with updated period value
if mode is NEXT_MODE
switch state to INPUT_STATE
end

Figure 4. Logic for WELCOME_STATE in FSM

<u>**Input State**</u>

Once the FSM enters the INPUT_STATE, the instructions for how Nucleotype operates are printed to the user's terminal. The program then calls the `getSentence()` function to generate a random sentence of 50 non-repeating words from a word bank and prints it to the user terminal in yellow using the function `printSentence()`.

Once the sentence has been printed on the screen, the program waits for the user to begin typing to start the timer. This feature ensures that the user does not have to frantically type immediately when the sentence is printed, and they instead get to preview the randomly generated sentence before starting the timer. Once the timer begins, the countdown is displayed on the LCD peripheral. When the user is typing, several handlers functions are utilized to ensure the typing experience is seamless (additional info. in the *Helper Function* section). All user input is read character by character and stored in an ArrayList data structure. Once the timer reaches the set time duration, the `beepBuzzer()` function is called to signal the typing test is done and the LCD is cleared. The FSM now unconditionally switches to the RESULT_STATE.



```
briannguyen — screen /dev/cu.usbmodem2103 115200 • SCREEN — 94×29
-------------
| TIME TEST |
-------------

1) Your goal is to type as many words as possible
2) Start typing to begin!
** CAUTION: you can only backpace within a word but not to a previous completed word

quare might number group she same at expert hold mean
you level play is school bottom bold be help aware
beard take robot could ready hair beat love can for
could he year off plan chaos board such cat blade
word people a write explain increase many take time benefit
```

Figure 5. Terminal screen capture of INPUT_STATE

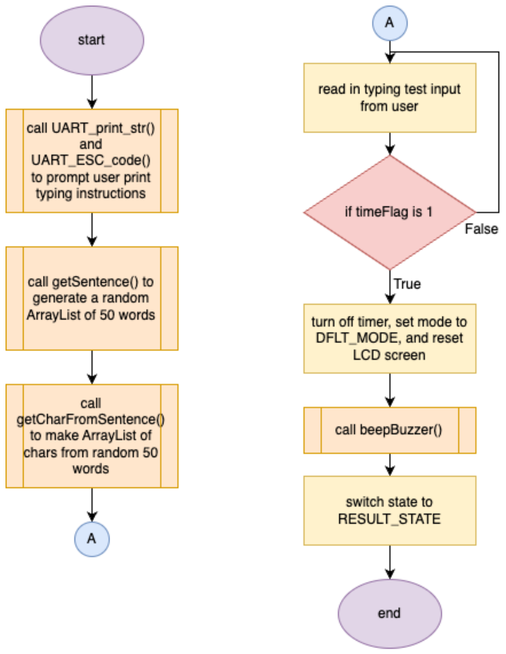
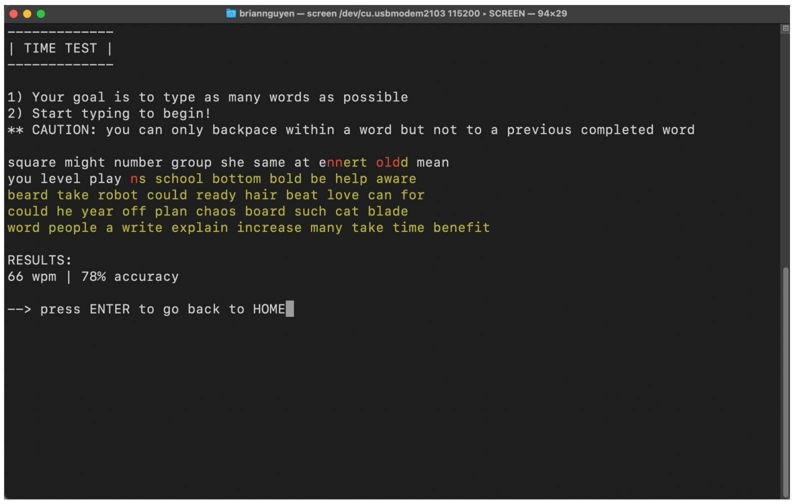

start
call UART_print_str() and UART_ESC_code() to prompt user print typing instructions
call getSentence() to generate a random ArrayList of 50 words
call getCharFromSentence() to make ArrayList of chars from random 50 words
A
A
read in typing test input from user
if timeFlag is 1
False
True
turn off timer, set mode to DFLT_MODE, and reset LCD screen
call beepBuzzer()
switch state to RESULT_STATE
end

Figure 6. Logic for INPUT_STATE in FSM

Result State

Once the FSM enters the RESULT_STATE, the `analyzeInput()` function is called and takes in the input ArrayList, the initial randomly generated sentence, and the timing mode as parameters. This function will compute the words-per-minute and typing accuracy of the user, and the results are printed on the screen (additional info. in the *Helper Function* section). In this state, all memory-allocated data from the use of each ArrayList are freed. Once the user acknowledges their results, they are prompted to press "ENTER" to switch back to the WELCOME_STATE for another try at Nucleotype. All global and register values are reset.



```
briannguyen — screen /dev/cu.usbmodem2103 115200 • SCREEN — 94×29
-------------
| TIME TEST |
-------------

1) Your goal is to type as many words as possible
2) Start typing to begin!
** CAUTION: you can only backpace within a word but not to a previous completed word

square might number group she same at ennert oldd mean
you level play ns school bottom bold be help aware
beard take robot could ready hair beat love can for
could he year off plan chaos board such cat blade
word people a write explain increase many take time benefit

RESULTS:
66 wpm | 78% accuracy

--> press ENTER to go back to HOME
```

Figure 7. Terminal screen capture of RESULT_STATE

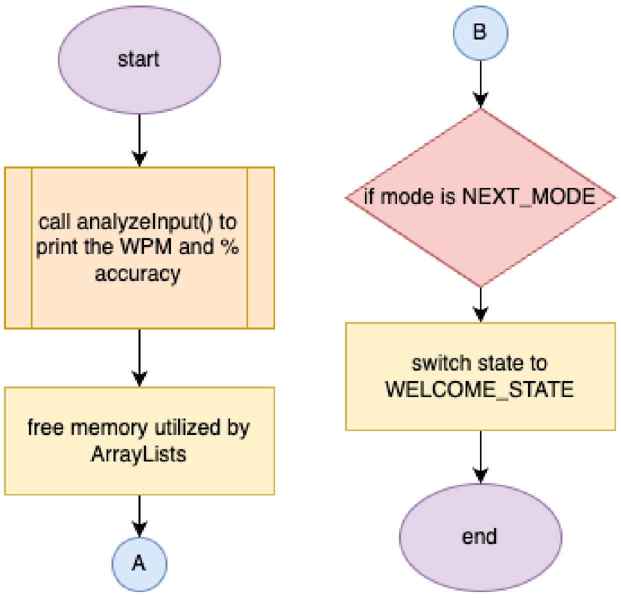

start
call analyzeInput() to print the WPM and % accuracy
free memory utilized by ArrayLists
A
B
if mode is NEXT_MODE
switch state to WELCOME_STATE
end

Figure 8. Logic for RESULT_STATE in FSM

Timers & Interrupts

In this program, a general-purpose timer TIM2 was used to time the user input, control the buzzer, and refresh the LCD screen. ARR interrupts were used to toggle a buzzer tone when each timed typing test finished. The ARR value varied for the 10, 15, and 30-second timing modes and was calculated using the following equation:

$$ARR = \frac{DESIRED\ TIME}{CLOCK\ PERIOD} - 1 = \frac{t}{(1/(24*10^{6}\ Hz)} - 1,$$ for t = 10, 15, or 30 seconds

For 10 seconds: $ARR = \frac{10\ s}{(1/(24*10^{6}\ Hz)} - 1 = 240000000$

For 15 seconds: $ARR = \frac{15\ s}{(1/(24*10^{6}\ Hz)} - 1 = 360000000$

For 30 seconds: $ARR = \frac{30\ s}{(1/(24*10^{6}\ Hz)} - 1 = 720000000$

CCR1 interrupts were also used to control when the LCD display refreshed so show a new count value. The count displayed was decremented and displayed every second. The CCR1 increment value used for each timing mode was determined by the following equation:

$CCR1 \mathrel{+}= \frac{ARR}{t} - 1$, where ARR is specific to the timing mode as discussed above and t is the time limit of 10, 15, or 30 seconds.

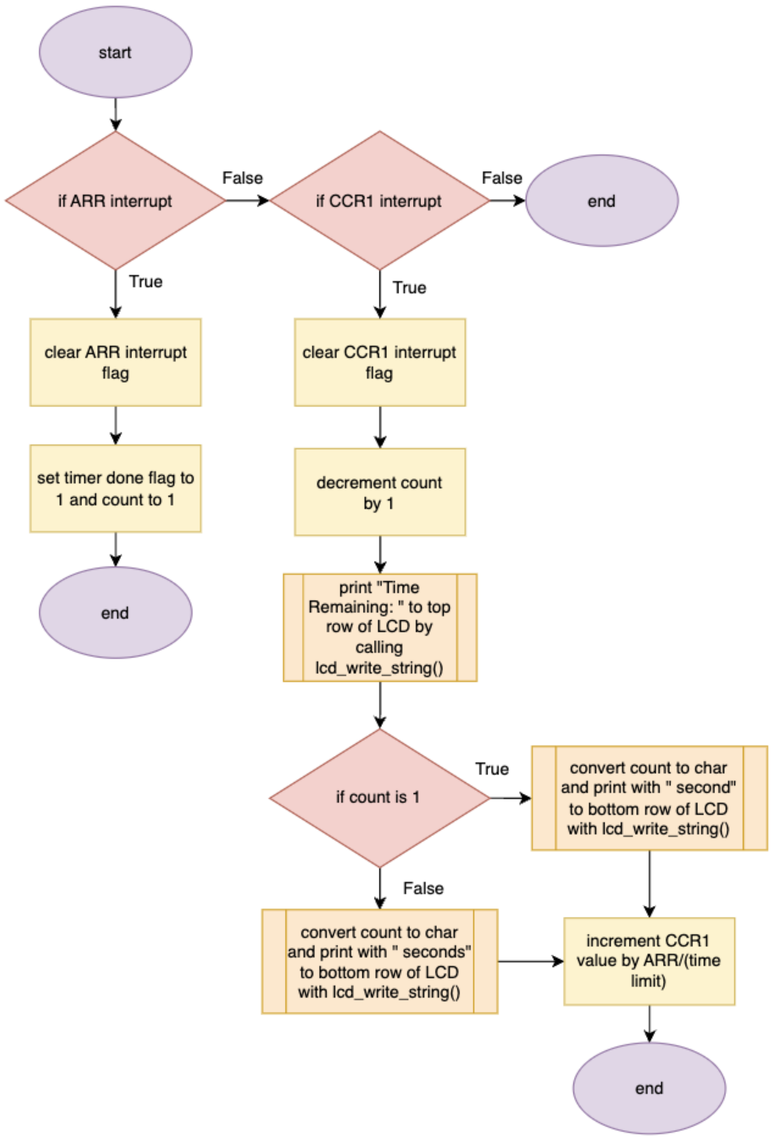

start
if ARR interrupt
False
if CCR1 interrupt
False
end
True
True
clear ARR interrupt flag
clear CCR1 interrupt flag
set timer done flag to 1 and count to 1
decrement count by 1
end
print "Time Remaining: " to top row of LCD by calling lcd_write_string()
if count is 1
True
convert count to char and print with " second" to bottom row of LCD with lcd_write_string()
False
convert count to char and print with " seconds" to bottom row of LCD with lcd_write_string()
increment CCR1 value by ARR/(time limit)
end

Figure 9. TIM2 Handler used to control timer end flag (ARR) and LCD count update every second (CCR1)

UART

In this program, UART uses RS-232 to establish a serial connection between the NUCLEO board and a virtual COM port connected to the debugger via a USB cable. The COM port can be interacted with using the VT100 serial terminal emulator which gives the user full use of the device's connected keyboard. This program takes advantage of UART receive-interrupts to capture keypresses as input data and compares this data character by character to the generated list of words that has to be typed for each typing test. UART was initialized with a USARTDIV value calculated from the following equation:

$$USARTDIV = \frac{FCLK}{Tx/Rx\ Baud\ (oversampling\ by\ 16)} = \frac{24\times10^{6}\ Hz}{115.2\times10^{3} bits/sec} \cong 208 = 0xD0$$

In addition to capturing input keypresses, the UART handler is implemented to be most optimal for user experience via text alignment and error checking. More specifically, when the typing test begins, the user input is printed in white over the initially printed yellow sentence instead of on another line. This enables the user to type more efficiently as they do not have to split their attention between two bodies of text and struggle to keep track of where their cursor is located separately. With this overlapping text feature, the UART handler ensures that all user input text is aligned and overlapping the original text by applying a unique sequence of code whenever the "BACKSPACE" or "SPACE" key is pressed.

When the "BACKSPACE" key is pressed, the handler shifts its cursor position to the left and writes the character from the initial randomly generated sentence in yellow. Additionally, the input ArrayList also shrinks dynamically as a reflection of the most recently inputted character being omitted. Finally, the user's cursor is shifted back once more to emulate the action of deleting a character. The backspace feature is limited to deleting words within the current word being typed and is unable to backspace to previously completed words.

When the "SPACE" key is pressed, the handler calls the `moveCursor()` function that takes in the current user's cursor position and the original random sentence as input. This function will return how much the user's cursor should shift to the right to be aligned with the next available word. This feature is important because if the user were to type fewer characters than in the actual word and move on, the following user input will be misaligned. Additionally, the handler ensures that the "SPACE" key is not pressed more than once to mess up the text alignment.

For the error-checking feature, the handler actively compares the user input to the prompted sentence character by character, and if the characters did not match, the handler would print the specific user-inputted character in red.

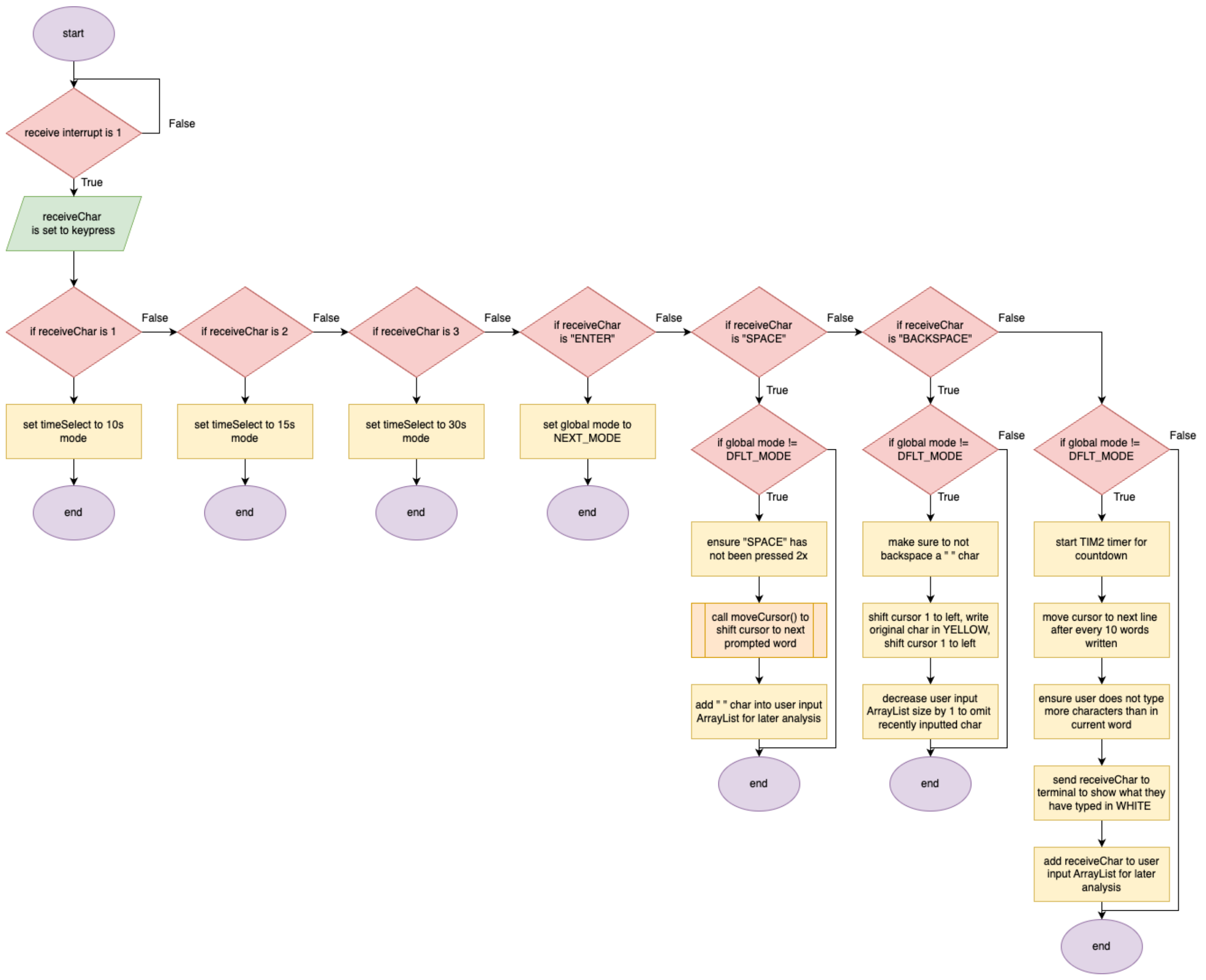

start
receive interrupt is 1
False
True
receiveChar is set to keypress
if receiveChar is 1
if receiveChar is 2
if receiveChar is 3
if receiveChar is "ENTER"
if receiveChar is "SPACE"
if receiveChar is "BACKSPACE"
set timeSelect to 10s mode
set timeSelect to 15s mode
set timeSelect to 30s mode
set global mode to NEXT_MODE
if global mode != DFLT_MODE
ensure "SPACE" has not been pressed 2x
call moveCursor() to shift cursor to next prompted word
add " " char into user input ArrayList for later analysis
make sure to not backspace a " " char
shift cursor 1 to left, write original char in YELLOW, shift cursor 1 to left
decrease user input ArrayList size by 1 to omit recently inputted char
start TIM2 timer for countdown
move cursor to next line after every 10 words written
ensure user does not type more characters than in current word
send receiveChar to terminal to show what they have typed in WHITE
add receiveChar to user input ArrayList for later analysis
end

Figure 10. USART2 Handler conditional checks for optimized User Experience and switching modes

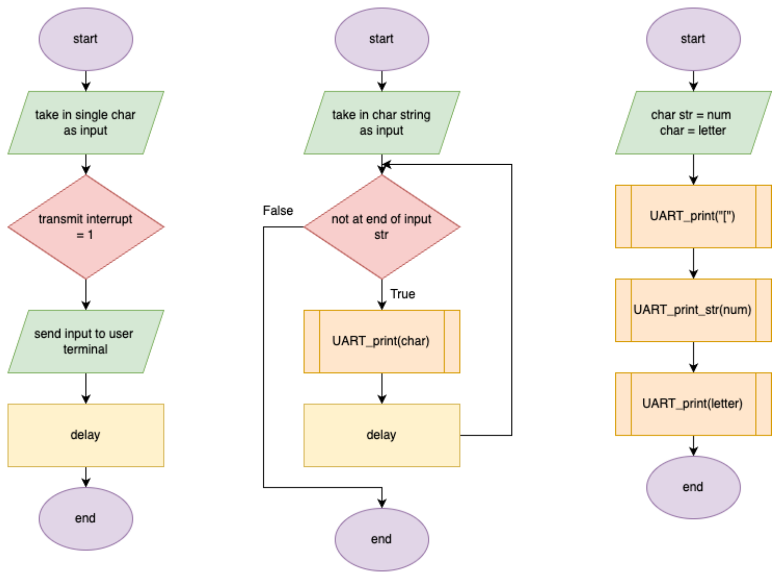


Figure 11. Flowcharts for UART_print(), UART_print_str(), and UART_ESC_code respectively

LCD

In this device, the LCD serves a single function: to display the time remaining for the user to type words. The LCD works by sending the higher 4-bit nibble and then the lower 4-bit nibble of an 8-bit character in between LCD pulses performed with `lcd_pulse_en()` all inside the `lcd_command()` function. The address to send the command is set and printing occurs in the `lcd_write_char()` function. A string of characters can be printed using the `lcd_write_string()` function. When the program is first run, the screen is initialized to be blank. It stays blank as the user selects a mode and is about to begin typing. Once the user types the first letter in their typing test, the LCD displays "Time Remaining" on the top row and the count of seconds that are remaining on the bottom row. The bottom row continues to be updated as CCR1 interrupts are hit and it stops being updated once ARR has been hit at zero seconds, signifying the end of the typing test. At this point, the bottom row is cleared entirely until a new test is started.

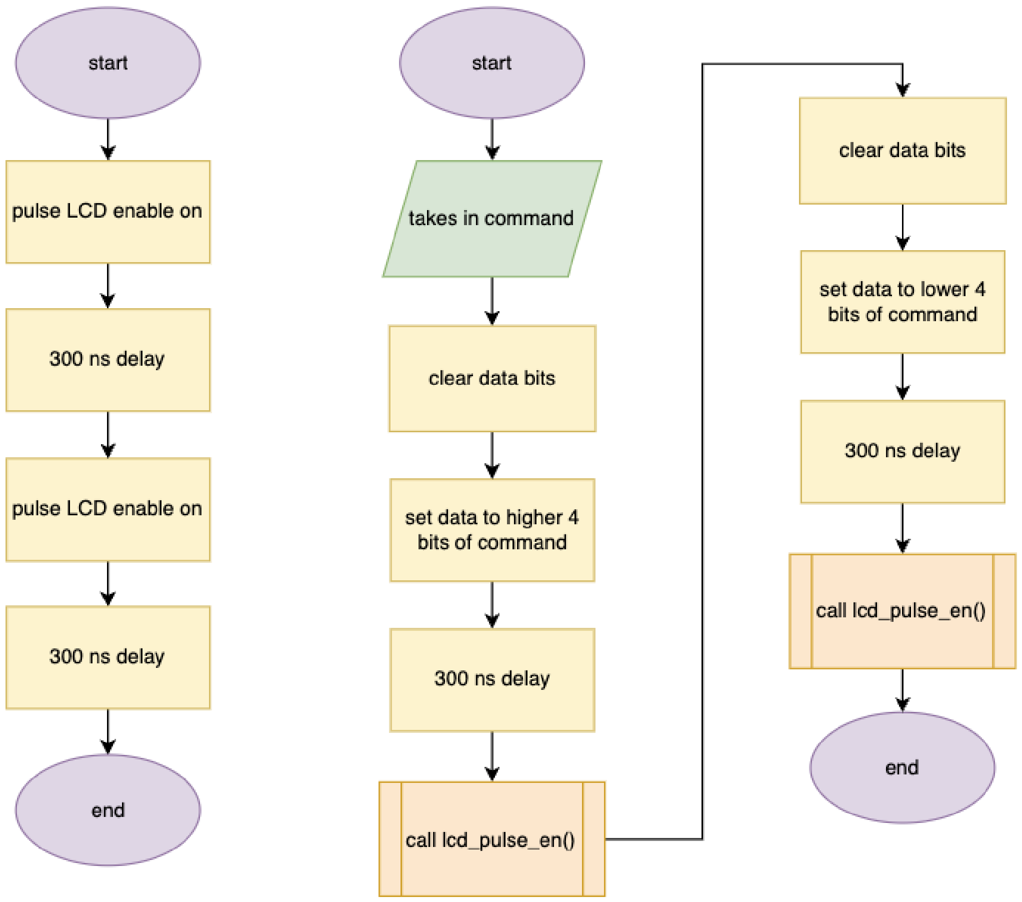


Figure 12. Flowcharts for lcd_pulse_en() and lcd_command() respectively

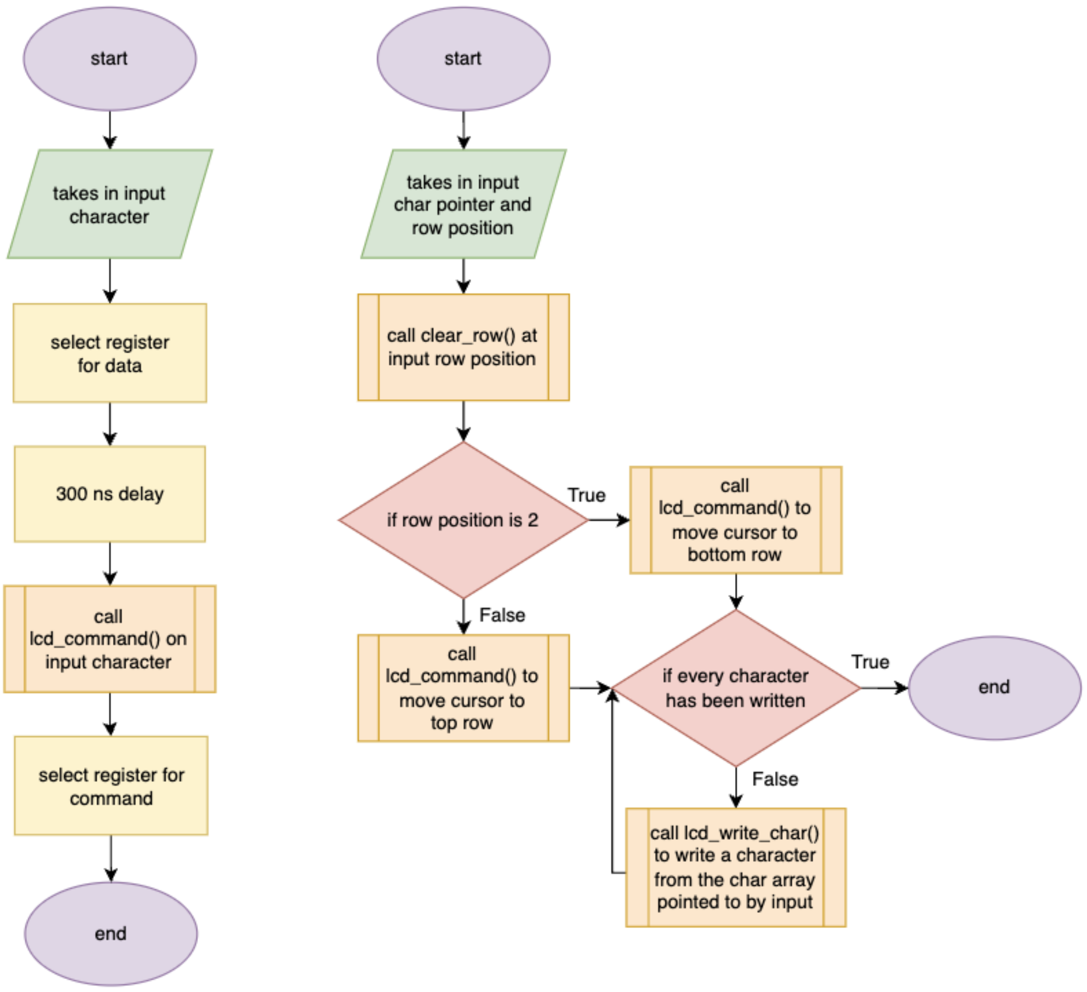

start
takes in input character
select register for data
300 ns delay
call lcd_command() on input character
select register for command
end
start
takes in input char pointer and row position
call clear_row() at input row position
if row position is 2
True
call lcd_command() to move cursor to bottom row
False
call lcd_command() to move cursor to top row
if every character has been written
True
end
False
call lcd_write_char() to write a character from the char array pointed to by input

Figure 13. Flowcharts for lcd_write_char and lcd_write_string respectively

<u>Buzzer</u>

In this device, an active buzzer is used to alert the user that their typing test has been finished. Since it is an active buzzer, it takes a 3.3 V DC source and uses internal electronics to create the buzzer tone. The buzzer is controlled through a single GPIO pin and has been configured with a `beepBuzzer()` function to trigger a series of five beeps every time the ARR value has been hit for each time mode. After the five beeps, the buzzer is turned off and does not activate again until another ARR interrupt occurs.

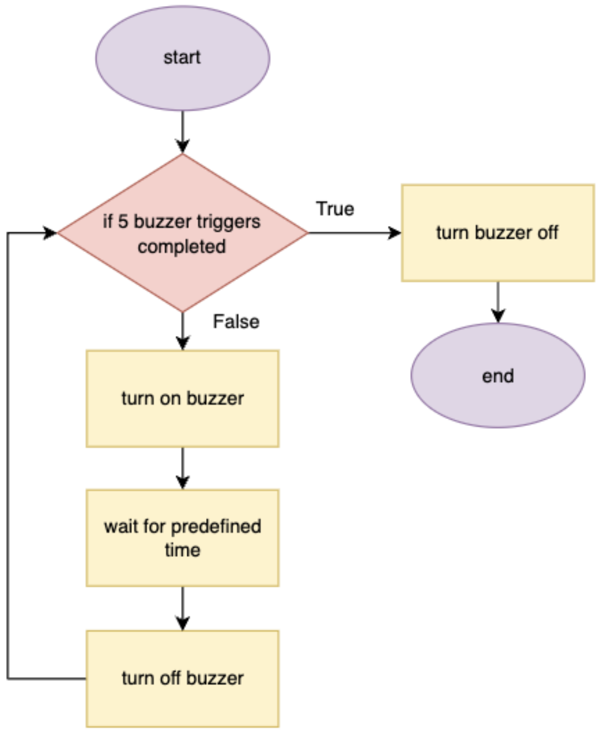


Figure 14. Logic for creating the buzzer tone when time is up

Helper Functions

`checked_malloc()`: This function checks to make sure `malloc()` can be called properly and is used when creating a new ArrayList structure.

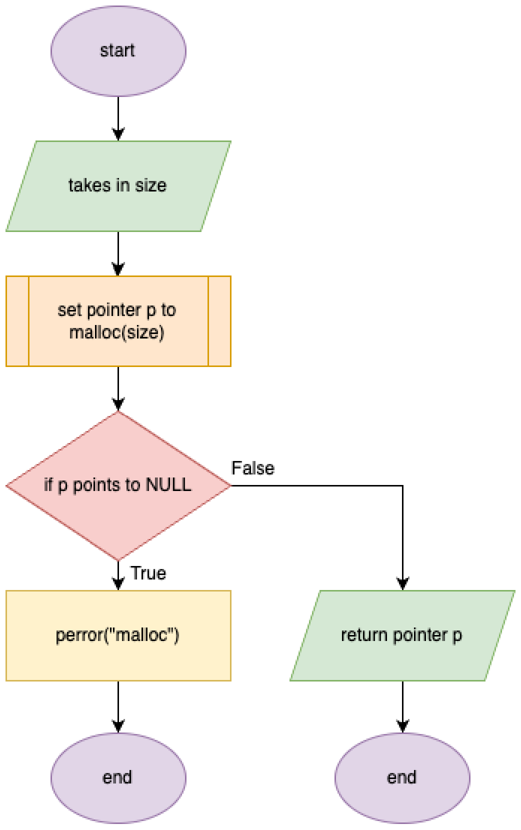


Figure 15. Logic for ensuring malloc does not result in NULL

`array_list_new():` This function creates a new ArrayList struct with size, capacity, and array attributes. This data structure is necessary for our project design because it is used to hold user input data, and the size of this data varies depending on how many words a user is able to type for each typing test. Having a dynamically allocated list allows us to be much more efficient with the space used by user input because we can add or remove space in the ArrayList as necessary.

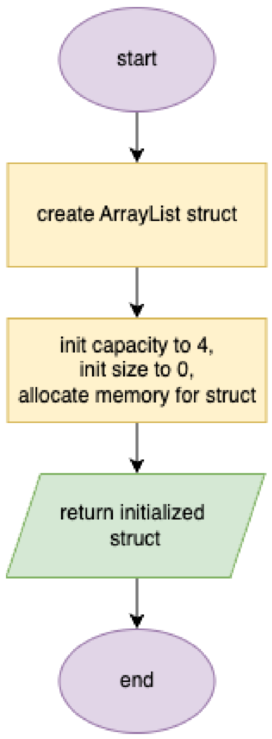


Figure 16. Logic for how the ArrayList struct is initialized

`array_list_add_to_end()`: This function increases the size attribute of the Arraylist struct and adds the input string to the last element of the ArrayList. This function is used to fill a newly created ArrayList with words from the word bank.

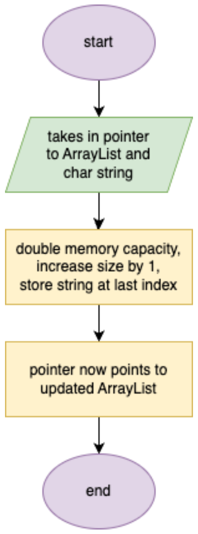

start
takes in pointer to ArrayList and char string
double memory capacity, increase size by 1, store string at last index
pointer now points to updated ArrayList
end

Figure 17. Logic for how the ArrayList dynamically grows when given an input

`randomNumber()`: This function takes a minimum and maximum value to create a range and then uses the `rand()` library function to generate a random number within that range. In our program, the size of the word bank determines the range of indices that can be randomly generated to create each 50-word ArrayList.

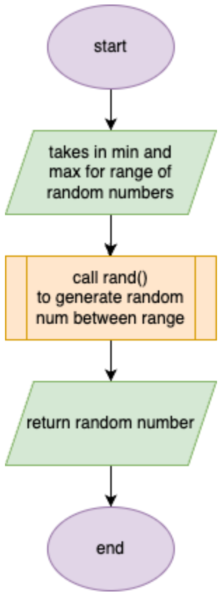

start
takes in min and max for range of random numbers
call rand() to generate random num between range
return random number
end

Figure 18. Logic for how a number is randomly generated given a min and max

`getSentence()`: This function controls the generation of the series of words that the user must type for each typing test. At the beginning of the function, a word bank consisting of 100 words is initialized. A series of 50 random numbers is then generated and added to an array to serve as the indices for extracting random words. Another array is used to determine whether or not a random number has been previously chosen, and if it has not, it sets the element at the index denoted by the random number to a 1. Finally, the array of random numbers is used to grab random numbers from the word bank and add them to an ArrayList structure that is returned.

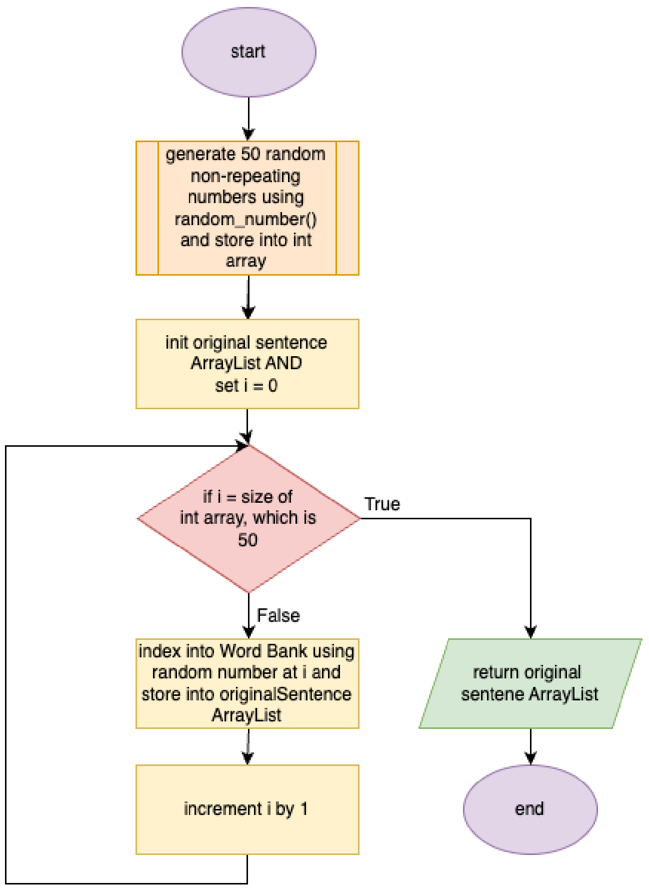

start
generate 50 random non-repeating numbers using random_number() and store into int array
init original sentence ArrayList AND set i = 0
if i = size of int array, which is 50
True
False
index into Word Bank using random number at i and store into originalSentence ArrayList
increment i by 1
return original sentene ArrayList
end

Figure 19. Logic for how prompted sentences are randomly generated and stored

`printSentence()`: This function prints out every word stored in the ArrayList returned by `getSentence()` so that the user can see what needs to be typed. The function only prints 10 words per line to prevent the user from getting overwhelmed by too many words on one line.

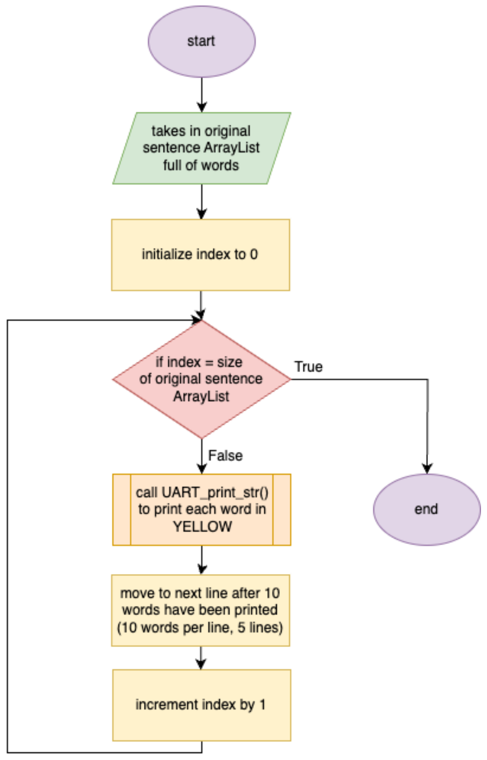

start
takes in original sentence ArrayList full of words
initialize index to 0
if index = size of original sentence ArrayList
True
False
call UART_print_str() to print each word in YELLOW
move to next line after 10 words have been printed (10 words per line, 5 lines)
increment index by 1
end

Figure 20. Simple logic to display the prompted sentence to the user

`getCharFromSentence()`: This function takes in the ArrayList returned by `getSentence()` and converts it into an ArrayList that stores all the characters in all the words that were in the original sentence. This returned ArrayList is used in the FSM as a reference to compare with the user input, which reads a single character at a time.

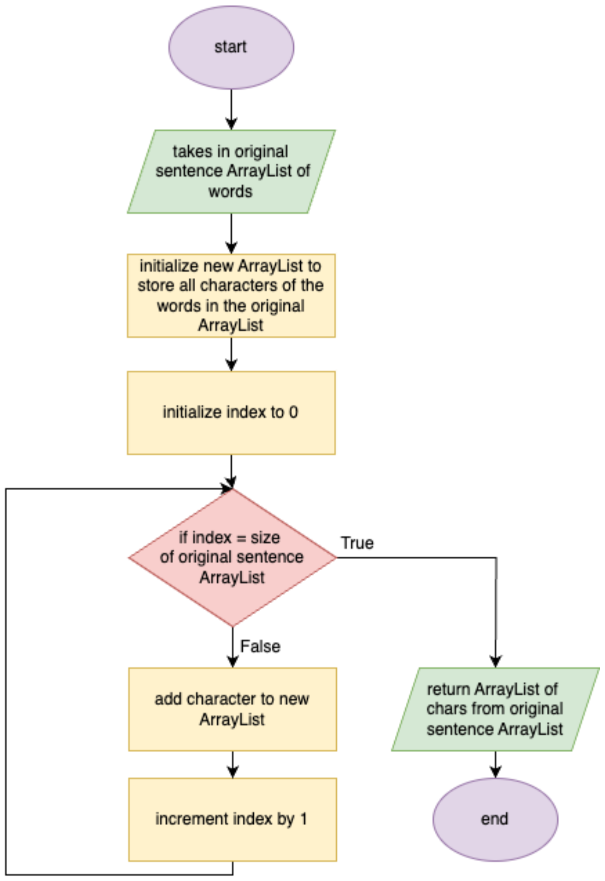

start
takes in original sentence ArrayList of words
initialize new ArrayList to store all characters of the words in the original ArrayList
initialize index to 0
if index = size of original sentence ArrayList
True
False
add character to new ArrayList
increment index by 1
return ArrayList of chars from original sentence ArrayList
end

Figure 21. Simple logic to convert ArrayList of words into ArrayList of characters

`moveCursor()`: This function takes in the current user's cursor position and the original random sentence as input. It returns how much the user's cursor should shift to the right to be aligned with the next available word.

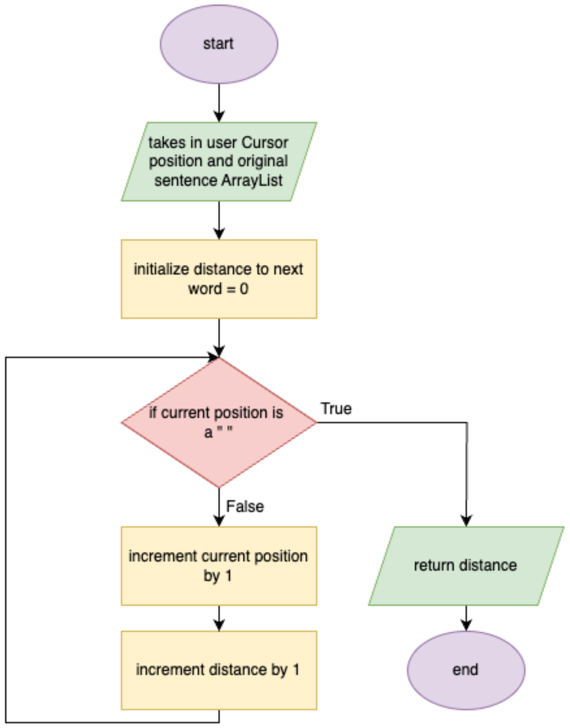

start
takes in user Cursor position and original sentence ArrayList
initialize distance to next word = 0
if current position is a " "
True
False
increment current position by 1
increment distance by 1
return distance
end

Figure 22. Logic to find the next prompted word and return new cursor position

`analyzeInput()`: This function takes in the input character ArrayList, the originally prompted sentence as an ArrayList of character strings, the originally prompted sentence as an ArrayList of characters, and the time mode. From the input character ArrayList, the function groups the characters into words by looking for a space and stores the word into a new ArrayList. The function then compares the user-inputted word to the same indexed word in the original sentence ArrayList and increments the number of valid words typed whenever the two words match. From this valid word count, the words per minute are scaled according to the time mode parameter, as shown below.

$$\begin{aligned}
Time\ Mode &= 10s \rightarrow WPM = validCount * 6 \\
Time\ Mode &= 15s \rightarrow WPM = validCount * 4 \\
Time\ Mode &= 30s \rightarrow WPM = validCount * 2
\end{aligned}$$

The user typing accuracy is calculated by finding the number of user-inputted characters that match at the same index of the original prompted ArrayList of characters and incrementing a valid character count variable. This function then finds the ratio of the valid character count to the total number of characters inputted and scales the value by 100. An example of when the valid character count is 78 and the total input character count is 96 is shown below:

$$\% = \frac{valid\ character\ count}{total\ character\ count} * 100$$

$$\% = \frac{78}{96} * 100$$

$$\% = 81\%$$

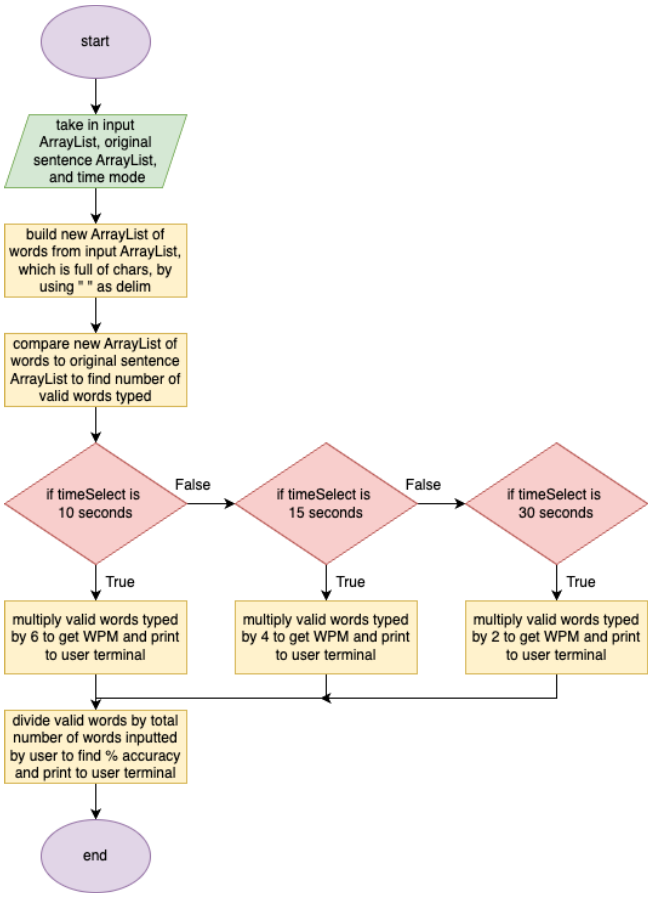

start
take in input ArrayList, original sentence ArrayList, and time mode
build new ArrayList of words from input ArrayList, which is full of chars, by using " " as delim
compare new ArrayList of words to original sentence ArrayList to find number of valid words typed
if timeSelect is 10 seconds
False
if timeSelect is 15 seconds
False
if timeSelect is 30 seconds
True
multiply valid words typed by 6 to get WPM and print to user terminal
True
multiply valid words typed by 4 to get WPM and print to user terminal
True
multiply valid words typed by 2 to get WPM and print to user terminal
divide valid words by total number of words inputted by user to find % accuracy and print to user terminal
end

Figure 23. Logic to determine Words Per Minute and % Accuracy for User Input

Power Calculations

For our device, power usage is considered for powering the STM32-L476RG microcontroller in its RUN state, on-board peripherals like the TIM2 timer and USART2, and external peripherals like the buzzer and LCD.

Power Consumption:

The average current per test was calculated by finding the current consumption per cycle and dividing it by the total cycle time. The upper limit current consumption for each peripheral was determined by looking at its datasheet. It was assumed that the STM32 was in RUN mode using the main voltage regulator. For each timed test, the cycle current, cycle time, and average current consumption were calculated using the following equations, where t represents the amount of time per test:

$$I_{cycle,\,t} = (I_{STM(RUN)} + I_{TIM2} + I_{UART} + I_{LCD})(t) + (I_{Buzzer} * t_{Buzzer})$$

$$\Rightarrow I_{cycle,\,t} = ((112\mu A/MHz * 24MHz) + (6.8\mu A/MHz * 24MHz) + (1.4\mu A/MHz * 24) + 0.25\,mA)(t) + (35mA * 0.008333s * 5\,buzzer\ beeps)$$

$$T_{cycle,\,t} = t + T_{Buzzer}$$

$$\Rightarrow T_{cycle,\,t} = t + (35mA * 0.008333s * 5\,buzzer\ beeps)$$

$$I_{AVG,\,t} = (I_{cycle,\,t} / T_{cycle,\,t})$$

After average current was found for each typing test, it was multiplied by the 3.3V supply voltage that comes directly from the STM32 microcontroller. Total power was then calculated using the following standard equation:

$$P_{AVG,\,t} = I_{AVG,\,t} * V_{Supply}$$

The results of these calculations can be seen in the table below.

Time Setting (s)	I_{AVG} (mA)	P_{AVG} (mW)
10	3.2670	10.78
15	3.2231	10.64
30	3.1790	10.49

Table 2. Power Consumption Calculations

The current draw and power consumption decrease as typing time increases due to there being more time for the current and power to be distributed throughout.

Battery Life:

For the theoretical case that this device would have to rely on battery power, the total battery life was calculated if only one type of timed test was done continuously for as long as possible. For this situation, a DC-DC converter module would need to be used in conjunction with three 1.5V D-cell batteries in series to supply the needed 3.3V supply voltage to every component. According to the datasheets, the selected DC-DC converter has a 93 percent conversion efficiency and the selected 1.5V D-cell batteries have a capacity rating of approximately 20,000 mAh for current discharges hovering around 25 mA. The desired current out of the batteries in series and the battery capacity estimate were calculated using the following equations:

$$P_{in,\,t} * Efficiency \;=\; P_{out}$$
$$\Rightarrow V_{Battery} * I_{Battery,\,t} * Effciency = I_{AVG,\,t} * V_{Supply}$$
$$\Rightarrow I_{Battery,\,t} = (I_{AVG,\,t} * 3.3V)/(4.5V * 0.93)$$

$$Battery\,Life \;=\; Capacity \,/\, I_{Battery,\,t}$$
$$\Rightarrow Battery\,Life \;=\; 20000mAh \,/\, I_{Battery,\,t}$$

The results of these calculations can be seen in the table below.

Time Setting (s)	$I_{Battery}$ (mA)	Battery Life (hours)	Battery Life (days)	Tests Completed
10	2.5761	7763.59	323.48	2794892
15	2.5415	7869.33	327.89	1888639
30	2.5067	7978.50	332.44	957420

Table 3. Battery Life Calculations

User Manual

Welcome to Nucleotype! This project is operated directly through the VT1000 serial terminal emulator, so no interaction with the hardware is necessary once the program is running. Once the device is connected to power, the reset button on the STM32 board can be pressed to load the program into the terminal.

At startup, you will be asked to choose from the following selections for timing mode by pressing the associated key:

[1] → 10 seconds
[2] → 15 seconds
[3] → 30 seconds

This allows you to select how long they want their typing test to run for. Once the key associated with the timing mode has been pressed, you will be asked to press enter to confirm that you want to go to the test. A randomly generated set of 50 words will then be displayed in yellow text to indicate to you what you will have to type during their test. As indicated in the on-screen instructions, the goal is to type as many words as possible during the selected time, and once the test is complete, your typing speed in words per minute and typing accuracy as a percentage will be displayed. It is also noted that if you make a typing error, you will only be able to backspace within the word you are currently typing.

To start the test, simply start typing the first word. Once this happens, the LCD display connected to the STM32 board will start displaying a count of how many seconds are left before the typing test is up. This display updates every second, and once the test ends, the bottom row of the LCD is cleared. Once you have started typing, you should keep typing until time ends to try to get the highest typing speed you can. An important note about accuracy is that you can only get 100% accuracy if you type every word correctly and stop typing a new word before the timer goes off. If you are in the middle of a word when the timer goes off, the system will count that word as being spelled incorrectly.

When time is up, a buzzer tone will go off to let you know. You will also not be able to type any more words on the screen. Your results will be displayed below the generated words. If you want to start a new test, all you need to do is press enter on the results page and you will be taken back to the screen where you can select the timing mode.

Appendix

Code:

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "main.h"
#include "helper.h"
#include "usart.h"
#include "lcd.h"
#define CLEAR 0
#define POS_DF -1
#define CURSOR_DF 8
void SystemClock_Config(void);
typedef enum {
       WELCOME_STATE,
       INPUT_STATE,
       RESULT_STATE,
} state_var_type;
state_var_type state = WELCOME_STATE;
uint16_t mode = DFLT_MODE;
uint16_t timeSelect = CLEAR;
uint32_t period = 0;
uint16_t pos = POS_DF;
uint16_t cursorPos = CURSOR_DF;
uint16_t spaceFlg = CLEAR;
uint16_t timeFlg = CLEAR;
uint16_t spaceNum = 0;
uint16_t wordsPerLine = 0;
uint16_t constCount;
uint16_t count;
char count_buff[10];
char seconds_string[9] = " seconds";
struct _arraylist* charList;
struct _arraylist* sentenceCharList;
int main(void)
{
       // STM32
       HAL_Init();
       SystemClock_Config();
       // UART
       UART_init();
       // TIM2
       TIM2_config();
       // Buzzer
       Buzzer_config();
       //LCD
       SysTick_Init();
       init_lcd_port();
```

```c
lcd_init();
// RNG
srand(time(NULL));  // Initialize random number generator with a time-based seed
struct _arraylist* currSentence;
while (1)
{
	switch(state){
	case WELCOME_STATE:
		pos = POS_DF;
		cursorPos = CURSOR_DF;
		timeSelect = CLEAR;
		TIM2->CNT = 0;
		timeFlg = CLEAR;
		wordsPerLine = 0;
		UART_ESC_code("37", 'm');		// white font
		UART_ESC_code("2", 'J');		// clear screen
		UART_ESC_code("", 'H');			// top left
		/*** print instructions ***/
		UART_print_str("-------------------------");
		UART_ESC_code("2;0", 'H');
		UART_print_str("| WELCOME to NUCLEOTYPE |");
		UART_ESC_code("3;0", 'H');
		UART_print_str("-------------------------");
		UART_ESC_code("5;0", 'H');
		UART_print_str("--> press 1, 2, or 3 to select timing mode.");
		UART_ESC_code("7;0", 'H');
		UART_print_str("[1] 10 seconds to type as many words as you can");
		UART_ESC_code("8;0", 'H');
		UART_print_str("[2] 15 seconds to type as many words as you can");
		UART_ESC_code("9;0", 'H');
		UART_print_str("[3] 30 seconds to type as many words as you can");
		/*** wait for time selection ***/
		period = 0;
		while(1){
			if (timeSelect != 0){
				if (timeSelect == TIME1){
					period = PERIOD1;
					count = 10;
					constCount = 10;
				}
				if (timeSelect == TIME2){
					period = PERIOD2;
					count = 15;
					constCount = 15;
				}
				if (timeSelect == TIME3){
					period = PERIOD3;
					count = 30;
					constCount = 30;
				}
				char timeBuf[10];
				UART_ESC_code("11;0", 'H');
```

```
                        UART_print_str("Mode [");
                        UART_print_str(itoa(timeSelect, timeBuf, 10));
                        UART_print_str("] selected");
                        break;
                }
        }
        TIM2->ARR = period - 1;
        TIM2->CCR1 = period / 15;
        UART_ESC_code("13;0", 'H');
        UART_print_str("--> press ENTER to continue");
        /*** wait for mode select ***/
        while(1){
                if (mode != DFLT_MODE){
                        break;
                }
        }
        /*** switch to desired mode ***/
        state = INPUT_STATE;
break;
case INPUT_STATE:
        /*** print title and instructions ***/
        UART_ESC_code("2", 'J');            // clear screen
        UART_ESC_code("", 'H');             // top left
        UART_print_str("-------------");
        UART_ESC_code("2;0", 'H');
        UART_print_str("| TIME TEST |");
        UART_ESC_code("3;0", 'H');
        UART_print_str("-------------");
        UART_ESC_code("5;0", 'H');
        UART_print_str("1) Your goal is to type as many words as possible");
        currSentence = getSentence();
        sentenceCharList = getCharFromSentence(currSentence);
        /*** print sentence and get user input ***/
        UART_ESC_code("6;0", 'H');
        UART_print_str("2) Start typing to begin!");
        UART_ESC_code("7;0", 'H');
        UART_print_str("** CAUTION: you can only backpace within a word but
                  not to a previous completed word");
        UART_ESC_code("9;0", 'H');
        UART_ESC_code("33", 'm');           // yellow color
        printSentence(currSentence);        // print prompt sentence
        UART_ESC_code("37", 'm');           // white color
        UART_ESC_code("9;0", 'H');
        /*** store and analyze user input ***/
        charList = array_list_new();
        while(timeFlg != 1);
        TIM2->CR1 &= ~(TIM_CR1_CEN);        // stop timer
        mode = DFLT_MODE;                   // reset mode
        beepBuzzer();
        clear_row(2);
        state = RESULT_STATE;
break;
```

```
		case RESULT_STATE:
			UART_ESC_code("37", 'm');		// white color
			UART_ESC_code("15;0", 'H');
			UART_print_str("RESULTS:");
			UART_ESC_code("16;0", 'H');
			analyzeInput(charList, currSentence, timeSelect);
			UART_ESC_code("18;0", 'H');
			UART_print_str("--> press ENTER to go back to HOME");
			/*** wait for home input ***/
			while(1){
				if (mode == NEXT_MODE){
					break;
				}
			}
			mode = DFLT_MODE;			// reset mode
			/*** free memory ***/
			free(currSentence->arraylist);
			free(currSentence);
			free(sentenceCharList->arraylist);
			free(sentenceCharList);
			free(charList->arraylist);
			free(charList);
			/*** switch back to welcome ***/
			state = WELCOME_STATE;
		break;
		}
	}
}
void USART2_IRQHandler(void){
	if((USART2->ISR & USART_ISR_RXNE) != 0){
		char receive_char = USART2->RDR;
		if (receive_char == '1'){
			timeSelect = TIME1;
		}
		else if (receive_char == '2'){
			timeSelect = TIME2;
		}
		else if (receive_char == '3'){
			timeSelect = TIME3;
		}
		else if (receive_char == '\r'){
			mode = NEXT_MODE;
		}
		else if (receive_char == '\x7F'){
			if (mode != DFLT_MODE){
				char* letter = sentenceCharList->arraylist[pos];
				if (strcmp(letter, " ") != 0){
					UART_ESC_code("1", 'D');		// left 1
					UART_ESC_code("33", 'm');		// yellow font
					UART_print_str(letter);
					UART_ESC_code("37", 'm');		// white font
					UART_ESC_code("1", 'D');		// left 1
```

```
                        if (charList->size != 0){
                                charList->size -= 1;  // input charlist -= 1
                        }
                        pos -= 1;                     // shift pos back
                }
        }
}
else if (receive_char == ' '){
        if (mode != DFLT_MODE){
                // move cursor to next word
                if (spaceNum < 1){
                        char buf[10];
                        uint16_t tempPos = moveCursor(pos, sentenceCharList);
                        char* move = itoa(tempPos - pos, buf, 10);
                        UART_ESC_code(move, 'C');
                        pos = tempPos;
                        array_list_add_to_end(charList, " ");
                        spaceNum += 1;
                        wordsPerLine += 1;
                }
        }
}
else {
        if (mode != DFLT_MODE){
                // increment global pos for sentenceCharList
                pos += 1;
                spaceNum = 0;
                // start count one first keypress
                if (pos == 0){
                        TIM2->CR1 |= (TIM_CR1_CEN);          // start timer
                         lcd_write_string("Time Remaining:", 1);
                         lcd_write_string(strcat(itoa(count, count_buff, 10),
                                          seconds_string), 2);
                }
                if (wordsPerLine == 10){
                        cursorPos += 1;
                        char cursorBuf[10];
                        UART_ESC_code("", 'H');
                        // top left
                        UART_ESC_code(itoa(cursorPos, cursorBuf, 10), 'B');
                       // down X
                        wordsPerLine = 0;
                }
                // ensure no overtyping a word for misalignment
                if (sentenceCharList->arraylist[pos][0] != ' '){
                        // check if input char is incorrect
                        if (receive_char != sentenceCharList
                                             ->arraylist[pos][0]){
                                UART_ESC_code("31", 'm');    // red font
                        }
                        // output char typed in
                        USART2->TDR = receive_char;
```

```
                                    UART_ESC_code("37", 'm');           // white font
                                    // convert received char to string and add to arraylist
                                    char str[2];
                                    str[0] = receive_char;
                                    str[1] = '\0';
                                    array_list_add_to_end(charList, strdup(str));
                              }
                              else {
                                    pos -= 1;
                              }
                        }
                  }
            }
      }
void TIM2_IRQHandler(void) {
    // check status register for update event flag
       if (TIM2->SR & TIM_SR_UIF) {
              TIM2->SR &= ~(TIM_SR_UIF);           // clear flag
              timeFlg = 1;
              count = 1;
       }
       else if (TIM2->SR & TIM_SR_CC1IF)
       {
              TIM2->SR &= ~(TIM_SR_CC1IF);              // clear ccr interrupt flag
              count -= 1;
              lcd_write_string("Time Remaining:", 1);
              if (count == 1) {
                     lcd_write_string(strcat(itoa(count, count_buff, 10), " second"), 2);
                                                               // print count in seconds
              }
              else {
                     lcd_write_string(strcat(itoa(count, count_buff, 10), seconds_string),
                                                         2);    // print count in seconds
              }
              TIM2->CCR1 += (period / constCount) - 1;    // increm CCR by 1 second
       }
}
```

----

helper.h

```
#include <stddef.h>
#include "main.h"
#ifndef SRC_HELPER_H_
#define SRC_HELPER_H_
#define DFLT_MODE 0
#define NEXT_MODE 1
#define TIME1 1
#define TIME2 2
#define TIME3 3
```

```
#define NUM_WORDS 50
#define BANK_SIZE 100
#define PERIOD1 240000000
#define PERIOD2 360000000
#define PERIOD3 720000000
void* checked_malloc(size_t size);
struct _arraylist
{
    size_t size;     // size of the list
    size_t capacity; // max size of list
    char* *arraylist;
};
struct _arraylist* array_list_new();
void TIM2_config(void);
void Buzzer_config(void);
void beepBuzzer(void);
void array_list_add_to_end(struct _arraylist* input_list, char *str);
void analyzeInput(struct _arraylist* charList, struct _arraylist* originalSentence, int
timeSelect);
int random_number(int min_num, int max_num);
struct _arraylist* getSentence();
struct _arraylist* getCharFromSentence(struct _arraylist* sentence);
void printSentence(struct _arraylist* sentence);
uint16_t moveCursor(uint16_t pos, struct _arraylist* sentenceCharList);
#endif /* SRC_HELPER_H_ */
```

----

helper.c

```
#include "helper.h"
#include "usart.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define INTIAL_CAPACITY 4
#define WAIT_TIME 200000
/*
 * Function configures ARR and CCR1 interrupts for TIM2
 * Does not return
*/
void TIM2_config(void) {
       // Configure TIM2 to interrupt every 15 seconds
       RCC->APB1ENR1 |= (RCC_APB1ENR1_TIM2EN);             // enable TIM2
       TIM2->DIER |= (TIM_DIER_UIE | TIM_DIER_CC1IE);      // enable interrupt on
                                                             update and CCR1 event
       TIM2->SR &= ~(TIM_SR_UIF | TIM_SR_CC1IF);           // clear update and CCR1
                                                             interrupt flag
       TIM2->CCMR1 &= ~(TIM_CCMR1_CC1S);                   // set CCR mode for output
       // enable TIM2 interrupts in NVIC
       NVIC->ISER[0] = (1 << (TIM2_IRQn & 0x1F));
       __enable_irq();   // enable interrupts globally
}
/*
 * Function checks if malloc can safely be called
 * Does not return
*/
void* checked_malloc(size_t size){
    int *p;
    p = malloc(size);
    if (p == NULL){
        perror("malloc");
        exit(1);
    }
    return p;
}
/*
 * Function configures GPIO to be used to activate buzzer
 * Does not return
*/
void Buzzer_config(void) {
    // enable clock for GPIOA
    RCC->AHB2ENR = RCC_AHB2ENR_GPIOAEN;
    // Configure PC0 for buzzer
    GPIOA->MODER &= ~(GPIO_MODER_MODE0);                    // clear MODES 0
    GPIOA->MODER |= (1 << GPIO_MODER_MODE0_Pos);            // set MODES 0
    GPIOA->OTYPER &= ~(GPIO_OTYPER_OT0);                    // push-pull
    GPIOA->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED0);              // low speed
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPD0);                    // no pull up / pull down
}
/*
 * Function creates buzzer tone used when timer ends
 * Does not return
*/
void beepBuzzer(void){
       for(int i = 0; i < 5; i++) {
         GPIOA->ODR &= ~(GPIO_ODR_OD0);              // turn off
         for(int i = 0; i < WAIT_TIME; i++);         // wait
         GPIOA->ODR |= (GPIO_ODR_OD0);               // turn on
         for(int i = 0; i < WAIT_TIME; i++);         // wait
       }
       GPIOA->ODR &= ~(GPIO_ODR_OD0);                // turn off
}
/*
 * Function creates new arraylist with default attributes
 * Returns arraylist struct
*/
struct _arraylist* array_list_new()
{
    struct _arraylist* list = checked_malloc(sizeof(struct _arraylist));
    list->size = 0;
    list->capacity = INTIAL_CAPACITY;
    list->arraylist = calloc(INTIAL_CAPACITY, sizeof(char*));
    return list;
}
/*
 * Function adds new node to end of arraylist and increases size
 * Does not return
*/
void array_list_add_to_end(struct _arraylist* input_list, char *str){
    if(input_list->size == input_list->capacity){
        char* *arr_new = NULL;
        size_t new_size= input_list->capacity * 2;
// update capacity
        arr_new = realloc(input_list->arraylist, sizeof(char*) * new_size);
        input_list->arraylist = arr_new;
        input_list->capacity = new_size;
    }
    input_list->arraylist[input_list->size] = str;
// update element
    input_list->size += 1;
// update size
}
/*
 * Function calculates and prints speed and accuracy
 * Does not return
 */
void analyzeInput(struct _arraylist* charList, struct _arraylist* originalSentence, int
timeSelect){
       struct _arraylist* inputWordList = array_list_new();
       char tempWord[10];           // init temp word
       int charPos = 0;             // init char pos in word
       memset(tempWord, '\0', sizeof(tempWord));
       for (int i = 0; i < charList->size; i++){
              // space means end of word -> store
              if (charList->arraylist[i][0] == ' '){
                     array_list_add_to_end(inputWordList, strdup(tempWord));
                     memset(tempWord, '\0', sizeof(tempWord));
                     charPos = 0;
              }
              // get the last word in char list
              else if (i == (charList->size - 1)){
                     tempWord[charPos] = charList->arraylist[i][0];
                     array_list_add_to_end(inputWordList, strdup(tempWord));
                     memset(tempWord, '\0', sizeof(tempWord));
              }
              // build word from chars
              else {
                     tempWord[charPos] = charList->arraylist[i][0];
                     charPos += 1;
              }
       }
       // determine number of input words are valid
       int validCount = 0;
       for (int i = 0; i < inputWordList->size; i++){
```

```
                if (strcmp(inputWordList->arraylist[i], originalSentence->arraylist[i]) ==
                                                                                        0){
                        validCount += 1;
                }
        }
        // scale to wpm and print
        int scaledValidCount = 0;
        // scale to 60 seconds
        if (timeSelect == TIME1){
                scaledValidCount = validCount * 6;
        }
        if (timeSelect == TIME2){
                scaledValidCount = validCount * 4;
        }
        if (timeSelect == TIME3){
                scaledValidCount = validCount * 2;
        }
        char tempBuf1[10];
        UART_print_str(itoa(scaledValidCount, tempBuf1, 10));
        UART_print_str(" wpm | ");
        // calculate accuracy and print
        int accuracy = (validCount * 100.0) / inputWordList->size;
        char accBuf[10];
        UART_print_str(itoa(accuracy, accBuf, 10));
        UART_print_str("% accuracy");
}
/*
 * Function generates a random number within min and max
 * Returns random number
*/
int random_number(int min_num, int max_num) {
        int min = min_num;
        int max = max_num;
        int randomNum;
        randomNum = (rand() % (max - min + 1)) + min;
        return randomNum;
}
/*
 * Function generates a sentence of 50 non-repeating words
 * Returns ArrayList containing words for the prompted sentence
*/
struct _arraylist* getSentence() {
        struct _arraylist* final_sentence = array_list_new();
        char word_bank[BANK_SIZE][9] =
              {{"are"}, {"my"}, {"is"}, {"will"}, {"be"},
                {"can"}, {"what"}, {"for"}, {"might"}, {"need"},
                {"take"}, {"year"}, {"play"}, {"off"}, {"such"},
                {"head"}, {"house"}, {"number"}, {"sick"}, {"through"},
                {"she"}, {"mine"}, {"those"}, {"tasty"}, {"you"},
                {"mild"}, {"school"}, {"mean"}, {"a"}, {"look"},
                {"he"}, {"dead"}, {"people"}, {"more"}, {"expert"},
                {"chaos"}, {"come"}, {"same"}, {"help"}, {"get"},
                {"nature"}, {"problem"}, {"think"}, {"group"}, {"love"},
                {"benefit"}, {"lake"}, {"pure"}, {"heat"}, {"even"},
                {"future"}, {"square"}, {"level"}, {"could"}, {"plan"},
                {"world"}, {"lie"}, {"robot"}, {"bet"}, {"hand"},
                {"cottage"}, {"back"}, {"write"}, {"blade"}, {"hold"},
                {"bottom"}, {"could"}, {"major"}, {"now"}, {"beard"},
                {"where"}, {"your"}, {"at"}, {"many"}, {"into"},
                {"point"}, {"beat"}, {"consider"}, {"take"}, {"bold"},
                {"music"}, {"explain"}, {"bottle"}, {"chair"}, {"board"},
                {"scream"}, {"cat"}, {"smile"}, {"ready"}, {"hair"},
                {"major"}, {"around"}, {"light"}, {"time"}, {"word"},
                {"increase"}, {"feel"}, {"shoe"}, {"bake"}, {"aware"}};
        int used_array[BANK_SIZE];
        int random_array[NUM_WORDS];
        int random_num, count = 0;
        memset(used_array, 0, sizeof(used_array));             // initialize each element to 0
        while (count < NUM_WORDS) {
                random_num = random_number(0, BANK_SIZE - 1);  // generate random number
                if (!used_array[random_num]) {                 // check if number used before
                        random_array[count] = random_num;      // add num  to array
                        used_array[random_num] = 1;            // mark num as used
                        count++;                               // increase random num count
                }
        }
        // for each random number index, add associated word to final sentence arraylist
        for (int index = 0; index < NUM_WORDS; index++) {
                array_list_add_to_end(final_sentence,
strdup(word_bank[random_array[index]]));
        }
        return final_sentence;
}
/*
 * Function converts ArrayList of words into ArrayList of chars
 * Returns ArrayList of chars from original prompted sentence
*/
struct _arraylist* getCharFromSentence(struct _arraylist* sentence){
        struct _arraylist* sentenceCharList = array_list_new();
        for (int i = 0; i < sentence->size; i++){
                int length = strlen(sentence->arraylist[i]);
                for (int j = 0; j < length; j++){
                        char ptr[2];
                        ptr[0] = sentence->arraylist[i][j];
                        ptr[1] = '\0';
                        if (j == length - 1){
                                array_list_add_to_end(sentenceCharList, strdup(ptr));
                                array_list_add_to_end(sentenceCharList, " ");
                        }
                        else {
                                array_list_add_to_end(sentenceCharList, strdup(ptr));
                        }
                }
        }
```

```
       return sentenceCharList;
}
/*
 * Function prints the prompted sentence (10 words per line)
 * Does not return
*/
void printSentence(struct _arraylist* sentence) {
    char temp_count[5];
    for (int word = 0; word < sentence->size; word++) {
        if ((word % 10) == 0) {
              int temp_num = (word / 10) + 8;             // 8 is offset for formatting
            UART_ESC_code("", 'H');
            UART_ESC_code(itoa(temp_num, temp_count, 10), 'B'); // new line every 10 words
        }
        UART_print_str(sentence->arraylist[word]);
        UART_print(' ');
    }
}
/*
 * Function finds where the next word is in the prompted sentence
 * Returns amount cursor needs to shift to the right
*/
uint16_t moveCursor(uint16_t pos, struct _arraylist* sentenceCharList){
       for (int i = pos; i < sentenceCharList->size; i++){
              if (strcmp(sentenceCharList->arraylist[i], " ") == 0){
                     return i;
              }
       }
       return -1;
}
```

----

usart.h

```
#ifndef SRC_USART_H_
#define SRC_USART_H_
void UART_init(void);
void UART_print(char data);
void UART_print_str(char *str);
void UART_ESC_code(char *num, char letter);
#endif /* SRC_USART_H_ */
```

----

usart.c

```
--------------------------------------------------------------------------------
```

```
#include "string.h"
#include "main.h"
#define DELAY 1000
#define BRR_CLEAR 0xFFFF
#define BRR_SET       0xD0
#define OUT_PORT GPIOA
void UART_init(void){
       // enable clock
       RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;
       // set mode to alternate function [2 is b'10] for alternate function mode
       OUT_PORT->MODER &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3);
       OUT_PORT->MODER |= (2 << GPIO_MODER_MODE2_Pos | 2 << GPIO_MODER_MODE3_Pos);
       // establish ports A2, A3
       OUT_PORT->AFR[0] &= ~(GPIO_AFRL_AFSEL2 | GPIO_AFRL_AFSEL3);
       // shift 7 because AF7
       OUT_PORT->AFR[0] |= (7 << GPIO_AFRL_AFSEL2_Pos | 7 << GPIO_AFRL_AFSEL3_Pos);
       // set to fast speed where 3 is b'11 for fast speed
       OUT_PORT->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED2 | GPIO_OSPEEDR_OSPEED3);
       OUT_PORT->OSPEEDR |= (3 << GPIO_OSPEEDR_OSPEED2_Pos | 3 << GPIO_OSPEEDR_OSPEED3_Pos);
       RCC->APB1ENR1 |= (RCC_APB1ENR1_USART2EN); // enable clock
       USART2->CR1 &= ~(USART_CR1_OVER8);
       USART2->BRR &= ~(BRR_CLEAR);
       USART2->BRR |= (BRR_SET);           // set baud rate to 0xD0
       USART2->CR1 &= ~(USART_CR1_M);      // set 01 for word length of 8
       USART2->CR2 &= ~(USART_CR2_STOP);   // set the  STOP bit to 0 for 1 stop
       USART2->CR1 |= (USART_CR1_UE);      // set the enable to 1
       USART2->CR1 |= (USART_CR1_TE);      // set the transmitter enable to 1
       USART2->CR1 |= (USART_CR1_RE);      // set the receiver enable to 1
       USART2->CR1 |= (USART_CR1_RXNEIE);  // set the receiver interrupt enable to 1
       NVIC->ISER[1] |= (1 << (USART2_IRQn & 0x1F));
       __enable_irq();                                    // enable interrupt globally
}
void UART_print(char data){
       while ((USART2->ISR & USART_ISR_TXE) == 0){}       // if TXE is high, it is empty
       USART2->TDR = data;
       for (int i = 0; i < DELAY; i++);
}
void UART_print_str(char *str){
       for (int pos = 0; pos < strlen(str); pos++){
              UART_print(str[pos]);
       }
}
void UART_ESC_code(char *num, char letter){
       while ((USART2->ISR & USART_ISR_TXE) == 0){}       // if TXE is high, it is empty
       USART2->TDR = 0x1B;
       for (int i = 0; i < DELAY; i++);
       UART_print('[');
       UART_print_str(num);
       UART_print(letter);

}
```

----

lcd.h

```
// Code Attribution: Eric Huang, Douglas Liu
#include "main.h"
#ifndef SRC_LCD_H_
#define SRC_LCD_H_
#define LCD_PORT           (GPIOC)
#define LCD_PORT_EN        (RCC_AHB2ENR_GPIOCEN)
#define LCD_DATA           (0x0F) // PC0 - PC1 - PC2 - PC3
                                  // DB4 - DB5 - DB6 - DB7
#define LCD_RS             (0x10) // PC4
#define LCD_RW             (0x20) // PC5
#define LCD_EN             (0x40) // PC6
#define LCD_FUNC_SET_4BIT  (0x28)
#define LCD_SET_CURSOR     (0x10)
#define LCD_DISP_ON_BLINK  (0x0F)
#define LCD_ENTRY_SET      (0x06)
#define LCD_TOP_LINE       (0x80)
#define LCD_BOT_LINE       (0xC0)
#define LCD_CLEAR_DISP     (0x01)
void SysTick_Init(void);
void delay_us(const uint32_t time_us);
void init_lcd_port();
void lcd_init();
void lcd_pulse_en();
void lcd_4b_command(uint8_t command);
void lcd_command(uint8_t command);
void lcd_write_char(char letter);
void lcd_write_string(char *phrase, int rowPosition);
void clear_row(int rowPosition);
#endif /* SRC_LCD_H_ */
```

----

lcd.c

```
// Code Attribution: Eric Huang, Douglas Liu
#include "lcd.h"
#include "main.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Configures GPIO and RCC for port C (for LCD Display)
void init_lcd_port() {
       // Enable clock for port D
       RCC->AHB2ENR |= LCD_PORT_EN;
       // Clear mode registers for configuration
       LCD_PORT->MODER &= ~(GPIO_MODER_MODE0 | GPIO_MODER_MODE1 |
                            GPIO_MODER_MODE2 | GPIO_MODER_MODE3 |
                              GPIO_MODER_MODE4 | GPIO_MODER_MODE5 |
                              GPIO_MODER_MODE6);
        // Finish configuring ports as GP outputs
        LCD_PORT->MODER |= (GPIO_MODER_MODE0_0 | GPIO_MODER_MODE1_0 |
                              GPIO_MODER_MODE2_0 | GPIO_MODER_MODE3_0 |
                              GPIO_MODER_MODE4_0 | GPIO_MODER_MODE5_0 |
                              GPIO_MODER_MODE6_0);
        // Putting output bits into output push pull
        LCD_PORT->OTYPER &= ~(GPIO_OTYPER_OT0 | GPIO_OTYPER_OT1 |
                              GPIO_OTYPER_OT2 | GPIO_OTYPER_OT3 |
                              GPIO_OTYPER_OT4 | GPIO_OTYPER_OT5 |
                              GPIO_OTYPER_OT6);
        // No pull up pull down
        LCD_PORT->PUPDR &= ~(GPIO_PUPDR_PUPD0 | GPIO_PUPDR_PUPD1 |
                              GPIO_PUPDR_PUPD2 | GPIO_PUPDR_PUPD3 |
                              GPIO_PUPDR_PUPD4 | GPIO_PUPDR_PUPD5 |
                              GPIO_PUPDR_PUPD6);
        // Set output speed of registers to very fast
        LCD_PORT->OSPEEDR |= ((3 << GPIO_OSPEEDR_OSPEED0_Pos) |
                              (3 << GPIO_OSPEEDR_OSPEED1_Pos) |
                              (3 << GPIO_OSPEEDR_OSPEED2_Pos) |
                              (3 << GPIO_OSPEEDR_OSPEED3_Pos) |
                              (3 << GPIO_OSPEEDR_OSPEED4_Pos) |
                              (3 << GPIO_OSPEEDR_OSPEED5_Pos) |
                              (3 << GPIO_OSPEEDR_OSPEED6_Pos));
}
// Turns DISPLAY ON and SETS CURSOR.
void lcd_init() {
        delay_us(40000);                        // Power up wait > 40 ms
        for (int idx = 0; idx < 3; idx++) {     // wake up 1,2,3: DATA = 0011 XXXX
                lcd_4b_command(0x30);           // Send higher 4 bits of 8 bit cmd
                delay_us(200);                  // wait > 160 us
        }
        lcd_4b_command(0x20);              // Put 0x20 on output port
        delay_us(40);
        lcd_command(LCD_FUNC_SET_4BIT);    // FUNCTION SET : 4 bit/ 2 line
        delay_us(40);
        lcd_command(LCD_SET_CURSOR);       // SET CURSOR
        delay_us(40);
        lcd_command(LCD_DISP_ON_BLINK);    // DISP ON, cursor blinks
        delay_us(40);
        lcd_command(LCD_CLEAR_DISP);       // CLEAR Display
        delay_us(40);
        lcd_command(LCD_ENTRY_SET);        // ENTRY MODE SET
        delay_us(5000);                    // Extra long to finish initializing LCD
}
// Pulses the LCD ENABLE 0->1->0
void lcd_pulse_en() {
        LCD_PORT->ODR |= (LCD_EN);         // ENABLE HIGH
        delay_us(30);                      // Delay > 300 ns
        LCD_PORT->ODR &= ~(LCD_EN);        // ENABLE LOW
        delay_us(30);                      // Delay > 300 ns
```

```
}
// Sends the LCD only the higher 4 bit nibble of command
void lcd_4b_command(uint8_t command) {
        // Function primarily used for 'wake-up' 0x30 commands
        LCD_PORT->ODR &= ~(LCD_DATA);                   // clear DATA bits
        LCD_PORT->ODR |= (command >> 4);               // DATA = higher 4 bits of command
        delay_us(5);                                   // Delay > 5 ms
        lcd_pulse_en();
}
// Sends the higher 4 bit nibble first, then the lower of command
void lcd_command(uint8_t command) {
        LCD_PORT->ODR &= ~(LCD_DATA);                  // isolate cmd bits
        LCD_PORT->ODR |= ((command >> 4) & LCD_DATA); // HIGH shifted low
        delay_us(30);
        lcd_pulse_en();                                // Latch HIGH NIBBLE
        LCD_PORT->ODR &= ~(LCD_DATA);                  // Isolate cmd bits
        LCD_PORT->ODR |= (command & LCD_DATA);         // LOW nibble
        delay_us(30);
        lcd_pulse_en();                                // Latch LOW NIBBLE
}
// Writes a single character to LCD
void lcd_write_char(char letter) {
        LCD_PORT->ODR |= (LCD_RS);                     // RS = HI for data to address
        delay_us(30);
        lcd_command(letter);                           // character to print
        LCD_PORT->ODR &= ~(LCD_RS);                    // RS = LO
}
// Writes a string of characters (phrase) at a specified row of
void lcd_write_string(char *phrase, int rowPosition) {
        // Clear the specified row on LCD
        clear_row(rowPosition);
        // Set cursor to specified row on LCD
        if (rowPosition == 2)
                lcd_command(LCD_BOT_LINE);
        else
                lcd_command(LCD_TOP_LINE);
        // For every character in the string of characters
        // Output to LCD
        for (int idx = 0; phrase[idx] != '\0'; idx++) {
                lcd_write_char(phrase[idx]);
        }
}
// Clears only one specified row of the LCD (either 1 or 2)
void clear_row(int rowPosition) {
        // Set cursor to specified row on LCD
        if (rowPosition == 2)
                lcd_command(LCD_BOT_LINE);
        else
                lcd_command(LCD_TOP_LINE);
        // Write blank spaces to every 16 cells of that row of the LCD
        for (int idx = 0; idx < 16; idx++) {
                lcd_write_char(0b11111110);
        }
}
// Configures SysTick timer for use in delay function
void SysTick_Init(void) {
        // Enable SysTick timer & select CPU clock
        SysTick->CTRL |= (SysTick_CTRL_ENABLE_Msk | SysTick_CTRL_CLKSOURCE_Msk);
        // Disable interrupt
        SysTick->CTRL &= ~(SysTick_CTRL_TICKINT_Msk);
}
// Delay function to create more precise delay
void delay_us(const uint32_t time_us) {
        // set the counts for the specified delay
        SysTick->LOAD = (uint32_t)((time_us * (SystemCoreClock / 1000000)) - 1);
        // Clear timer count
        SysTick->VAL = 0;
        // Clear count flag
        SysTick->CTRL &= ~(SysTick_CTRL_COUNTFLAG_Msk);
        // Wait for flag
        while (!(SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk));
}
-----------------------------------------------------------------------------
```

Video Links:
Nucleotype Trailer (Pt. 1): https://youtube.com/shorts/svmQ5NqUuxg?feature=share
Nucleotype Trailer (Pt. 2): https://youtube.com/shorts/l8loY6glzSs?feature=share

References

[1] "50mA/200mA Inductor Built-in Step-Down 'micro DC/DC' Converters." Torex Semiconductor

[2] "Active buzzer (5V)," Addicore, https://www.addicore.com/Active-Buzzer-5V-p/ad146.htm (accessed Jun. 6, 2023).

[3] "Character Liquid Crystal Display Module (NHD-0216HZ-FSW-FBW-33V3C)." Newhaven Display International, Elgin, IL, Aug. 5, 2020

[4] "Energizer E95." Energizer

[5] J. Miodec, "Monkeytype: A minimalistic, customizable typing test," Monkeytype, http://www.monkeytype.com/ (accessed Jun. 6, 2023).

[6] P. Hummel, J. Gerfen, and T. Houalla, "STM32 Lab Manual (CPE 316)." .

[7] "STM32L476xx Datasheet." ST, Jun-2019.

[8] "STM32L4xxxx Reference Manual (RM)." ST, Jun-2021.

[9] "ST7066U Dot Matrix LCD Controller/Driver." Sitronix, May. 11, 2006