Lab 2: User Interface & Assembly Programming

ELEC4601

Jesse Levine (101185127)

Lab Section: A4E

1 Part 1: Display Setup

1.1 Pin Configurations

Pins	Configuration
PA6	GPIO_Output
PA7	GPIO_Output
PC7	GPIO_Output
PB4	GPIO_Output
PB5	GPIO_Output
PB6	GPIO_Output

Table 1: Pin Configurations for Part 1: Displaying names on LCD

1.2 LCD Display Output

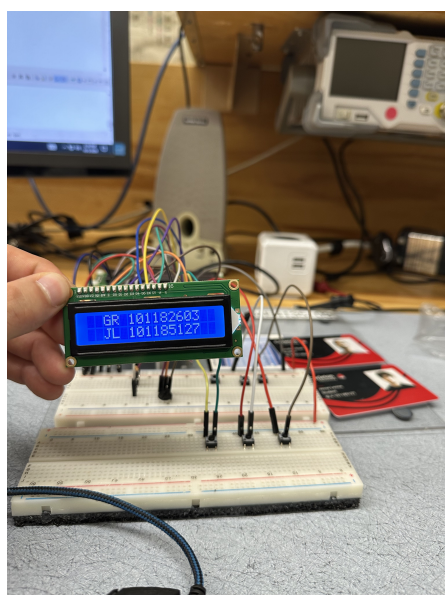

Figure 1: LCD Output for Part 1. The LCD displays our initials alongside our student numbers.

1.3 Code & Compiler

```
//Initialize ports and pins connected to the LCD
Lcd_PortType ports[] = { GPIOC, GPIOB, GPIOA, GPIOA };

Lcd_PinType pins[] = {GPIO_PIN_7, GPIO_PIN_6, GPIO_PIN_7, GPIO_PIN_6};

//Initialize lcd
Lcd_HandleTypeDef lcd;

lcd = Lcd_create(ports, pins, GPIOB, GPIO_PIN_5, GPIOB, GPIO_PIN_4, LCD_4_BIT_MODE);

//initialize the variables to be displayed on the LCD
char *CodeString = "JL 101185127";
char *CodeString_2 = "GR 101182603";

Lcd_cursor(&lcd, 0,2); //CodeString will be displayed on the first row beginning in the third cell
Lcd_string(&lcd, CodeString);

Lcd_cursor(&lcd, 1,2);  //CodeString_2 will be displayed on the second row beginning in the third cell
Lcd_string(&lcd, CodeString_2);
```

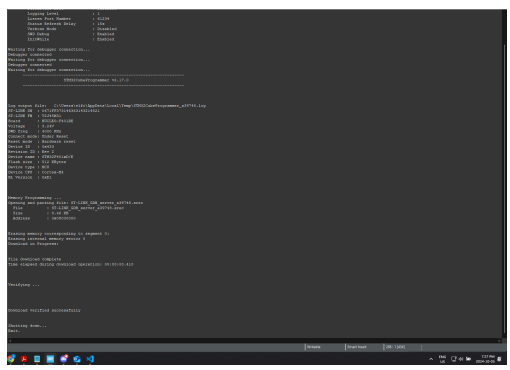

Figure 2: Compiler Output after Compiling & Running Part 1 Code. The screenshot was taken on my home computer hence the location of the Log Output File. This part was completed in the lab, only the screenshots are taken later.

2 Part 2: Input Keys Setup

2.1 Pin Configurations

Pins	Configuration
PA6	GPIO_Output
PA7	GPIO_Output
PC7	GPIO_Output
PB4	GPIO_Output
PB5	GPIO_Output
PB6	GPIO_Output
PC13	GPIO_Input
PA8	GPIO_Input
PA10	GPIO_Input

Table 2: Pin Configurations for Part 2. Two external buttons connected to PA8 and PA10 are used. The third button used is the on-board USER button connected to PC13.

2.2 LCD Display

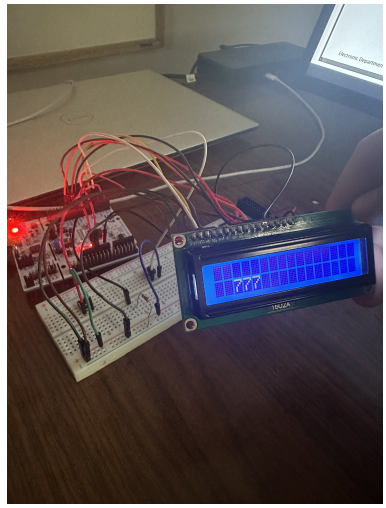

Figure 3: LCD Displaying the maximum number of 777. Three buttons are used, each connected to PA8, PA10 and PC13 respectively. One button increments the number by 1, another by 10 and the other by 100. If the number is 777, the next increment will restart at 0.

2.3 Code & Compiler

```
while (1){

Lcd_cursor(&lcd, 0,1);

if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_8)==GPIO_PIN_SET){
        x +=1;
        if( x > 777){
        x=1;
        }

        HAL_Delay(200);

}

if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)==0){
        x += 10;

        if( x > 777){
                x=1;
                }

}
```

```
if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_10)==1){
        x +=100;

        if( x > 777){
                x=1;
                }

        HAL_Delay(200);

}

if (x>777){
        x = 0;
        }

Lcd_cursor(&lcd, 1,2);

Lcd_int(&lcd, x);

HAL_Delay(200);
}
```

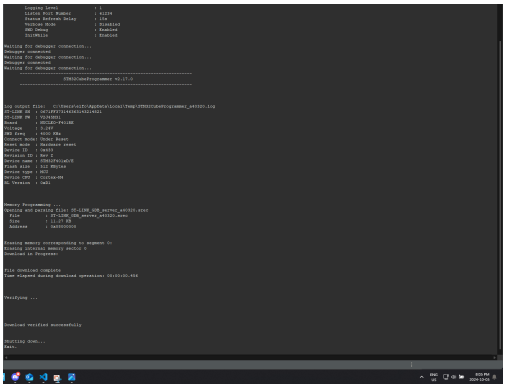

Figure 4: Compiler Output after Compiling & Running Part 2 Code. The screenshot was taken on my home computer hence the location of the Log Output File. This part was completed in the lab, only the screenshots are taken later.

3 Part 3: Morse Decoder

3.1 Pin Configurations

Pins	Configuration
PA6	GPIO_Output
PA7	GPIO_Output
PC7	GPIO_Output
PB4	GPIO_Output
PB5	GPIO_Output
PB6	GPIO_Output

Table 3: Pin Configurations for Part 3: Morse Decoder. The pin configurations for Part 3 are the same from Part 1. There is no need for external inputs such as buttons in this part.

3.2 LCD Output

Figure 5: LCD Output after decoding a Morse Code. Notice along with our names, there appears to be random artifacts and symbols. We were not sure why this was occuring; most likely a code issue. We will see below the code, however, no where should there be extra random symbols being decoded.

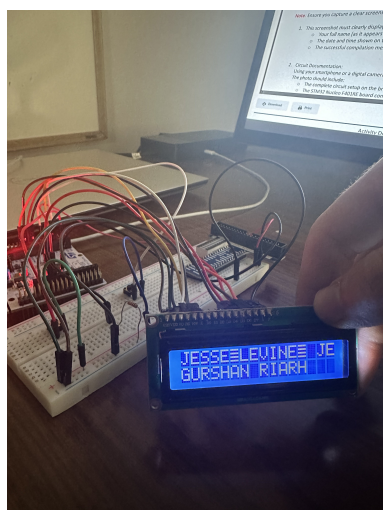

Figure 6: LCD Output after compiling same code a second time. For some reason, another compilation of the code produced less artifacts, than the first one in Figure 5

3.3 Code & Compiler

```
const char* morse_input_j = ".---";
const char* morse_input_e = ".";
const char* morse_input_s = "...";

const char* morse_input_L = ".-..";
const char* morse_input_v = "...-";
const char* morse_input_i = "..";
const char* morse_input_n = "-.";

const char* morse_input_g = "--.";
const char* morse_input_u = "..-";
const char* morse_input_r = ".-.";
const char* morse_input_h = "....";
const char* morse_input_a = ".-";

char translated_j = morse_to_char(morse_input_j);
char translated_e = morse_to_char(morse_input_e);
char translated_s = morse_to_char(morse_input_s);

char translated_L = morse_to_char(morse_input_L);
char translated_v = morse_to_char(morse_input_v);
char translated_i = morse_to_char(morse_input_i);
char translated_n = morse_to_char(morse_input_n);

char translated_g = morse_to_char(morse_input_g);
char translated_u = morse_to_char(morse_input_u);
char translated_r = morse_to_char(morse_input_r);
char translated_h = morse_to_char(morse_input_h);
char translated_a = morse_to_char(morse_input_a);

char CodeString[5];
CodeString[0] = translated_j;
CodeString[1] = translated_e;
CodeString[2] = translated_s;
CodeString[3] = translated_s;
CodeString[4] = translated_e;

char CodeString_1[6];
CodeString_1[0] = translated_L;
CodeString_1[1] = translated_e;
CodeString_1[2] = translated_v;
CodeString_1[3] = translated_i;
CodeString_1[4] = translated_n;
CodeString_1[5] = translated_e;

char CodeString2[7];
CodeString2[0] = translated_g;
CodeString2[1] = translated_u;
CodeString2[2] = translated_r;
CodeString2[3] = translated_s;
CodeString2[4] = translated_h;
CodeString2[5] = translated_a;
CodeString2[6] = translated_n;
```

```
char CodeString3[5];
CodeString3[0] = translated_r;
CodeString3[1] = translated_i;
CodeString3[2] = translated_a;
CodeString3[3] = translated_r;
CodeString3[4] = translated_h;



Lcd_PortType ports[] = { GPIOC, GPIOB, GPIOA, GPIOA };

Lcd_PinType pins[] = {GPIO_PIN_7, GPIO_PIN_6, GPIO_PIN_7, GPIO_PIN_6};

Lcd_HandleTypeDef lcd;

lcd = Lcd_create(ports, pins, GPIOB, GPIO_PIN_5, GPIOB, GPIO_PIN_4, LCD_4_BIT_MODE);

Lcd_cursor(&lcd, 0,0);
Lcd_string(&lcd, CodeString);

Lcd_cursor(&lcd, 0,6);
Lcd_string(&lcd, CodeString_1);



Lcd_cursor(&lcd, 1,0);
Lcd_string(&lcd, CodeString2);

Lcd_cursor(&lcd, 1,8);
Lcd_string(&lcd, CodeString3);
```

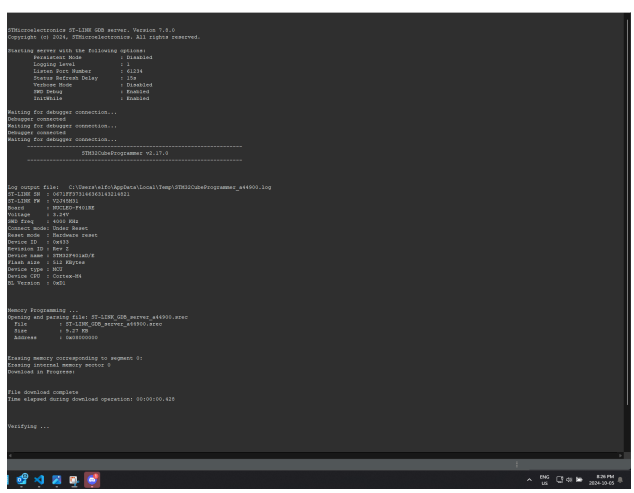

Figure 7: Compiler Output after Compiling & Running Part 3 Code. The screenshot was taken on my home computer hence the location of the Log Output File. This part was completed at home as well.

4 Circuit

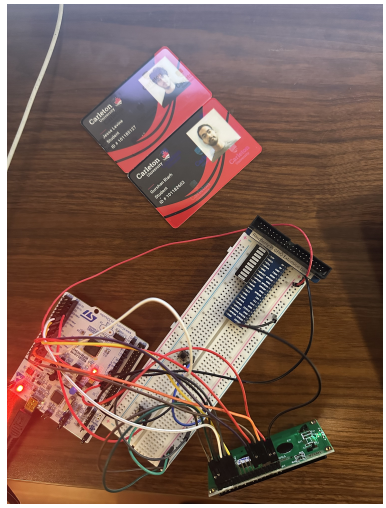

Figure 8: Full Circuit Implementation. Instead of a 10k potentiometer, I used a 220Ω resistor connected to the V0 pin on the LCD, the control the contrast of the LCD.

5 Explanation of morse_code_translator.s Assembly Code

At a high level, the assembly code defines a function, *morse_to_char*, that translates a string of Morse code into a single character using a lookup table, *morse_table*. In this case, the Morse code is comprised of dots and dashes.

The code can be explained in seven stages:

1. Function Setup:

```
.syntax unified
.thumb

.global morse_to_char
.type morse_to_char, %function
```

   Specify unified assembly and Thumb mode, declare *morse_to_char* as global function and instantiate it as an actual function

2. Function Prologue:

```
morse_to_char:
push {r4, r5, lr}
mov r4, r0
ldr r5, =morse_table
mov r0, #1
```

Save register r4 and r5 and return address to stack. Move pointer to morse code string into r4. Load address of morse code lookup table into r5 and initialize r0 with 1.

3. Morse Code Loop

```
morse_loop:
ldrb r1, [r4], #1
cmp r1, #0
beq translate
```

Load one byte into r1 and increment string pointer by 1. Compare loaded byte in r1 with 0 → comparing r1 with the end of the string. If theres a match, jump to *translate* block.

4. Encoding Morse Code

```
lsl r0, r0, #1

cmp r1, #'.'
beq morse_loop_continue
cmp r1, #'-'
bne invalid_code
add r0, r0, #1
```

Shift r0 left by 1 increment; loading space for next Morse code symbol. Compare byte with dot. If it's a dot continue to *more_lool_continue* block. If not compare with dash. If its not a dash or dot, branch to *invalid_code* block. If its a dash, add 1 to r0.

5. Continue Loop

```
morse_loop_continue:
b morse_loop
```

Repeats sequence for next character input.

6. Translation

```
translate:
cmp r0, #31
bhi invalid_code

ldrb r0, [r5, r0]
cmp r0, #0
beq invalid_code

pop {r4, r5, pc}
```

Compare encoded Morse with 31, which is the largest possible value. If r0 ¿ 31, branch to *invalid_code* block. If not, then load character from the table with r0 as index. Check if r0 is valid; if invalid branch to *invalid_code* block. Finally, restore registers and return from the function.

7. Invalid Code

```
invalid_code:
mov r0, #'?'
pop {r4, r5, pc}
```

Returns '?' in the case of invalid character and restores registers and returns to function.

6 Prelab

6.1 Register Usage in ARM R0-R15

Registers R0-R12 are general purpose registers and R13-R15 contain specialized roles.

However according to the ATPCS even the general purpose registers should be used "properly".

According to the ATPCS:

R0-R3 are to pass parameter values into routines and pass the values out.

R4-R11 are to hold values of a routine's local variables.

R12 is the intra-call scratch register: In procedure linkage, it used as such, however otherwise can be used as general purpose register.

R13 is the stack pointer; it points to the top of the stack.

R14 is the link register; it holds the return address when calling a subroutine.

R15 is the program counter; it keeps track of the next instruction to be executed.

6.2 Basic Arithmetic Instructions ADD,SUB, MUL

1. ADD

   ADD Rd, Rn, Rm

   The addition of Rn & Rm stored in Rd.

2. SUB

   SUB Rd, Rn, Rm

   The subtraction of Rn & Rm stored in Rd.

3. MUL

   MUL, Rd, Rn, Rm

   The multiplication of Rn & Rm stored in Rd.

6.3 Branching Instructions B, BL, BX

1. B

   Unconditionally branches to specified label/block of code.

2. BL

   Branches to specified label and stores the return address in the link register. Typically used for calling subroutines.

3. BX

   BX = Branch and Exchange. This branch operation branches to the address stored in a specified register and switches between ARM and Thumb modes depending on the least significant bit of the address.

6.4 Conditional Flags (N, Z, C, V)

1. Negative Flag

   Set when the result of an operation is a negative; if the most significant bit is a 1. Cleared if result is 0 or positive.

2. Zero Flag

   Set when the result of an operation is 0. Cleared otherwise

3. Carry Flag

   Set when an operation results in a carry-out from the most significant bit (if in the case of addition) or a borrow (if in this case of subtraction.) Cleared otherwise.

4. Overflow Flag

   Set when the result of an operation causes signed overflow; the result exceeds the range of the signed integer representation.

6.5 Short ARM Assembly Code

Question: "Write a short ARM assembly function that adds two numbers and returns the result. Explain each line of your code"

```
.syntax unified
.thumb

.global add_two_nums
.type add_two_nums, %function

add_two_nums:
```

```
push {lr}           @save link register to stack
add r0, r0, r1      @store the addition of the contents within r0 and r1 in r0
pop {lr}            @Restore link register from stack
bx lr               @return out of function using link register
```