# Lab 5 Report: Morse Code Translator

Athokshay Ashok

Group Member: Aryaman Pandya

Date: 12/5/2019

EE 14 - Microprocessor Architecture

Lab Section: Friday 10:30 am - 12:30 pm

Tufts University School of Engineering (EECS)

# Introduction

Morse code represents every character in the English language using three symbols: dot, dash, and space. Morse code symbols are sent at specific time intervals based on the symbol. A dot is one time unit, a dash is three time units, a space between letters is three time units, and a space between words is seven time units. However, for someone unfamiliar with Morse code, this can be a very complicated way of translating especially when they must constantly refer to the conversion table (shown in Appendix A). Moreover, a typical hardware converter (as seen in Figure 1) uses one button-like mechanism that is used for both dots and dashes; a small tap is a dot and a longer press is a dash. Since it can be easy to lose track of time units and make mistakes, we decided to implement a simplified version of a Morse Code translator that is beginner-friendly.

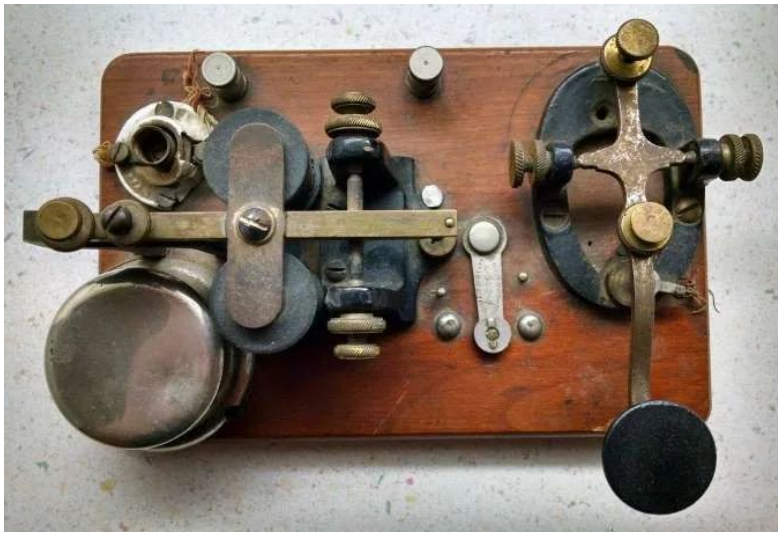


Figure 1: Traditional Morse Code Translator

Rather than using time as a metric, we looked at the conversion table and decided to standardize every character to be 5 symbols long, regardless of the symbols. For example, looking at Appendix A, A in morse code would be dot-dash-space-space-space. Since we are no longer using time to determine which symbol we are sending, we need separate inputs to send a dot, dash, and space. We also noticed that mistakes can easily be made. One wrong symbol can completely change the character, and so we decided to include an erase function that deletes the last sent symbol. In addition, the STM32L4 LCD display can only display 6 characters at a time, and for strings longer than that, we decided to implement a scrolling string function that can be called to view the entire string so far. Otherwise, if the translation is still in process, the LCD will display the six most recently translated characters by default.

# Experimental Setup and Discussion

Since the LCD display was required for this project, I began by using the LCD_Driver template from Lab 3. Having configured the LCD and implementing the display string and

moving string functions in Lab 3, I copied these functions from my Lab 3 code. To use delays, I also used the SysTick_Init, Delay, and SysTick_Handler functions from Lab 4.

The first step was to configure the joystick buttons as input. We had previously configured the center joystick button as input by checking the IDR in Lab 2 in assembly, so I used the same offsets to configure the port clock and moder in C. But the same logic did not work for the up, down, left, and right buttons. After a lot of trial and error and research, we realized that the center button was already configured as a pull down register, but the others were not. So to do this, I configured the PUPDR register for the appropriate pins along with setting the moders. This way, the device was able to recognize all 5 buttons as input.

The different controls are shown below in Figure 1.

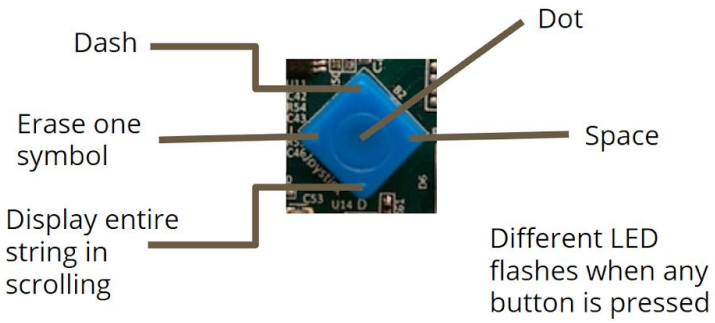



Figure 2: Joystick Controls

While testing the buttons, it was clear that the buttons could be easily mixed up since the joystick is relatively small. It also takes time for the chip to register a button press. So to indicate that a button has been pressed, we decided to flash a different LED each time a button was pressed, and to do this, we needed functions to turn on and off the green and red LEDs. The functions to turn on and off the green LED, as shown in Appendix C, configure the clock, moder, and set the ODR bit to 0 or 1 for off or on. To flash the LED, we simply turned on the LED, used a Delay of 500 to generate a 0.5 sec delay, and then turned off the LED.

To keep track of all the converted characters, we used a uint8_t* pointer to represent a string and a uint8_t integer to keep track of the length of the string. The biggest challenge was figuring out how to keep track of the inputted symbols. Our first thought was to use strings and string concatenation.

Starting with an empty string, for each:

1) Dot: append “d”
2) Dash: append “D”
3) Space: append “s”
4) Erase symbol: delete last character

But since we use uint8_t* pointers to represent strings and not char * (which is what the strcat function uses), this process did not work.
Instead, we used a global integer “morse” to do the symbol bookkeeping. Similar to the string concatenation, starting with morse = 0, for each:

1) Dot: morse = (morse * 10) + 1
2) Dash: morse = (morse * 10) + 3
3) Space: morse = morse * 10
4) Erase symbol: morse = morse / 10

For example, A is dot-dash-space-space-space. So morse = 13000.
Similarly, 1 is dot-dash-dash-dash-dash. So morse = 13333.

Having implemented a way of keeping track of the inputted symbols using numbers, we needed a way to translate these numbers into letters. This was done by translating each character from the conversion table in Appendix A into numbers and hard coding in each case. If morse equaled the integer representation of any of the hard coded characters, we appended that character to the string of all converted characters, incremented the length of the string, and reset morse to 0.

We then had to implement two types of displays: a default display, and a customized display. In the default display, the six most recently translated characters were displayed on the LCD. To do this, we malloced a new temporary string of length six, copied over the six most recent characters, and displayed them. If the string was less than 6 characters long, the string was displayed normally. The customized display, called by a joystick down press, displays the entire string so far in a scrolling fashion. To do this, we simply called the moving string function on the entire string and its length, and once the string finished displaying we reset the display to show the six most recent characters by using the method above.

# Results

Having implemented the translator, we started testing it by first going through the conversion chart and making sure that every character was properly translated individually. We then began testing the erase feature by intentionally making mistakes and erasing them. While testing, we found that our program would crash or glitch, and we were able to narrow it down to minor issues with the way we were looping or freeing the malloced strings. However, the program would still crash in a rare occasion, except when the joystick buttons were pressed

slowly. We then began testing longer strings to make sure that both the default and scrolling displays worked properly.

For example, the following string was generated using the following translations:
A (13000): Center-Up-Right-Right-Right
1 (13333): Center-Up-Up-Up-Up
B (31110): Up-Center-Center-Center-Right
2 (11333): Center-Center-Up-Up-Up
C (31310): Up-Center-Up-Center-Right
3 (11133): Center-Center-Center-Up-Up


Figure 3: Morse Code Translation Example

For obvious reasons, it is hard to demonstrate the exact conversions and scrolling string in a report.

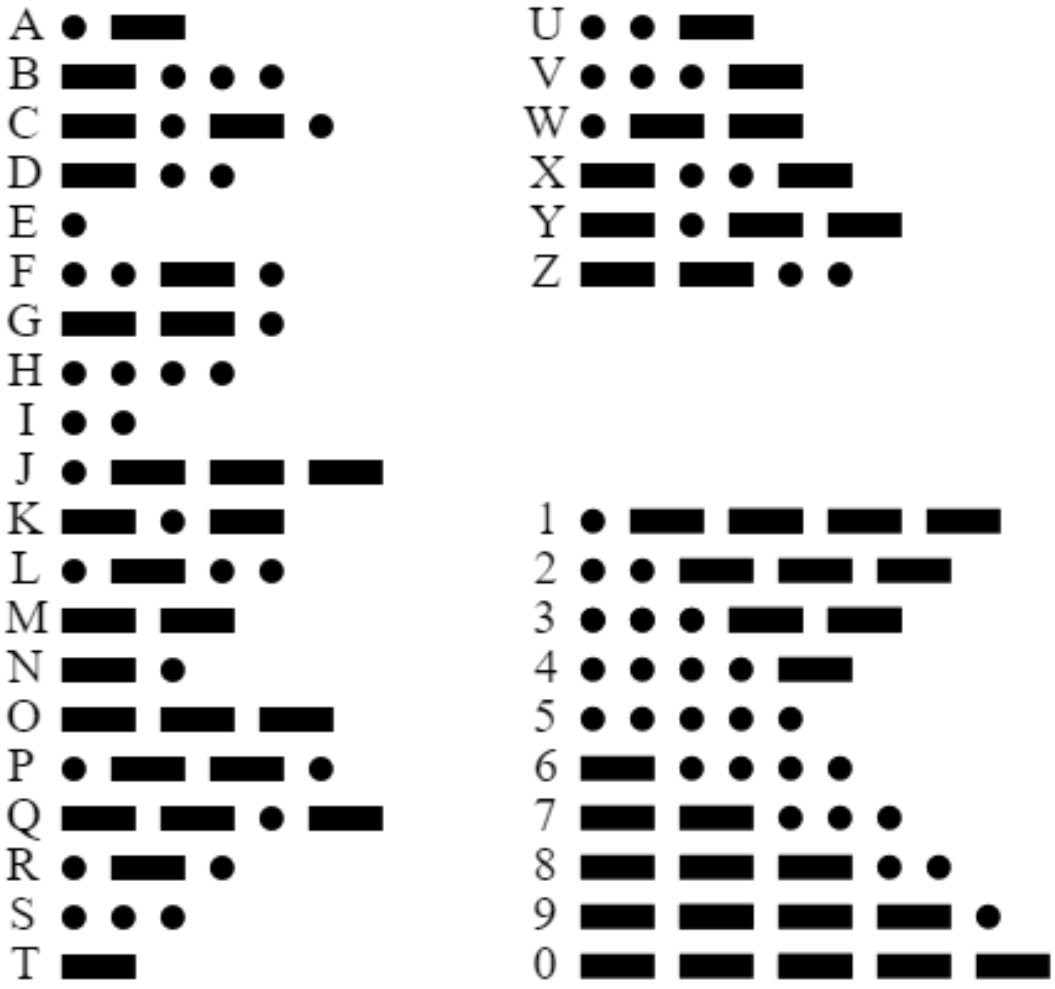
# Conclusion

We were able to successfully develop a simple morse code translator that a beginner could use by referring to the conversion table and the instructions for the controls. We also added additional features such as erasing a symbol and displaying the entire string so far. By getting rid of the time units, we were able to greatly simplify the translator and make it more efficient. The standard translator only includes on knob that is used for both spaces and dashes, which are conveyed through sound signals. These must then be interpreted by a person skilled in converting Morse code to English characters. Our translator includes an input mechanism and real time conversion of the symbols.

The biggest takeaways from this lab were the pull down configuration for the up, down, left, right buttons, and the manipulation of strings and integers to store values that we need. The microcontroller takes time to process inputs and display outputs, and this must be accounted for when using the translator. Going too fast could cause the program to crash and the display to wipe out. This unpredictable behavior, we believe, has something to do with the way the chip is compiling the program, or an unidentifiable memory issue with malloc that is interfering with the program and display.

# Appendix

Appendix A: Morse Code Translations



Appendix B: Joystick_Initialization

```
void Joystick_Initialization(void)
{
		RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;

	//Enable joystick middle: PA0
		GPIOA->MODER &= ~(0x03<<(2*0)) ;

	//Enable joystick left: PA1
		GPIOA->MODER &= ~(0x03<<(2*1)) ;
		GPIOA->PUPDR |= (0x02<<(2*1)) ;

	//Enable joystick right: PA2
		GPIOA->MODER &= ~(0x03<<(2*2)) ;
		GPIOA->PUPDR |= (0x02<<(2*2)) ;

	//Enable joystick up: PA3
		GPIOA->MODER &= ~(0x03<<(2*3)) ;
		GPIOA->PUPDR |= (0x02<<(2*3)) ;

		//Enable joystick down: PA5
		GPIOA->MODER &= ~(0x03<<(2*5)) ;
		GPIOA->PUPDR |= (0x02<<(2*5)) ;
}
```

Appendix C: Turn on and off green LED

```
void greenLED_On(void)
{
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOEEN;

	GPIOE->MODER &= ~(0x03<<(2*8)) ;

	GPIOE->MODER |= (1<<16);
	GPIOE->ODR |= GPIO_ODR_ODR_8;

}

void greenLED_Off(void)
{
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOEEN;

	GPIOE->MODER &= ~(0x03<<(2*8)) ;

	GPIOE->MODER &= ~(1<<16);
	GPIOE->ODR &= ~GPIO_ODR_ODR_8;
}
```

Appendix D: Translate_Morse

```
void Translate_Morse()
{

		if (morse == 13000)
		{
			string[curr_index] = 'A';
			curr_index ++;
			length ++;
			morse = 0;
		}

		else if (morse == 31110)
		{
			string[curr_index] = 'B';
			curr_index ++;
			length ++;
			morse = 0;
		}
		….
		// Actual code was too long
		// Repeat for C-Y

		else if (morse == 33110)
		{
			string[curr_index] = 'Z';
			curr_index ++;
			length ++;
			morse = 0;
		}

		// Numbers
		else if (morse == 13333)
		{
			string[curr_index] = '1';
			curr_index ++;
			length ++;
			morse = 0;
		}

		else if (morse == 11333)
		{
```

```
                        string[curr_index] = '2';
                        curr_index ++;
                        length ++;
                        morse = 0;
                }

                …
                //Repeat for 3-9
                else if (morse == 33333)
                {
                        string[curr_index] = '0';
                        curr_index ++;
                        length ++;
                        morse = 0;
                }

}
```

Appendix E: main()

```
uint8_t curr_index;
uint8_t * string;
uint8_t length;
uint16_t morse;

volatile uint32_t time;

int main(void)
{
		Joystick_Initialization();
		LCD_Initialization();
		SysTick_Initialize(4000);

		string = malloc (6 * sizeof(uint8_t));

		while (1)
		{
			uint16_t joystick_center = (((GPIOA->IDR) & (0x1 << 0)) >> 0);
				uint16_t joystick_left =   (((GPIOA->IDR) & (0x1 << 1)) >> 1);
				uint16_t joystick_right =  (((GPIOA->IDR) & (0x1 << 2)) >> 2) ;
				uint16_t joystick_up =     (((GPIOA->IDR) & (0x1 << 3)) >> 3) ;
				uint16_t joystick_down =   (((GPIOA->IDR) & (0x1 << 5)) >> 5) ;


				if (joystick_center == 1)
				{
						morse = (morse * 10) + 1;
						redLED_On();
						Delay(500);
						redLED_Off();
				}
				else if (joystick_left == 1)
				{
						morse = morse / 10;
						redLED_On();
						Delay(500);
						redLED_Off();
				}
				else if (joystick_right == 1)
```

```
{
                morse = morse * 10;
                redLED_On();
                Delay(500);
                redLED_Off();
}
else if (joystick_up == 1)
{
                greenLED_On();
                Delay(500);
                greenLED_Off();
                morse = (morse * 10) + 3;
}
else if (joystick_down == 1)
{
                greenLED_On();
                Delay(500);
                greenLED_Off();
                LCD_Clear();
                movingString(string, length);

                if (length > 6)
                {
                uint8_t * temp = malloc (6 * sizeof(uint8_t));
                        temp [5] = string [length - 1];
                        temp [4] = string [length - 2];
                        temp [3] = string [length - 3];
                        temp [2] = string [length - 4];
                        temp [1] = string [length - 5];
                        temp [0] = string [length - 6];
                        movingString(temp, 6);
                        free(temp);
                }
                else
                {
                                movingString(string, length);
                }
}


if (morse >= 10000)
{
        Translate_Morse();
```

```
                                        if (length > 6)
                                        {
                                                uint8_t * temp = malloc (6 * sizeof(uint8_t));
                                                        temp [5] = string [length - 1];
                                                        temp [4] = string [length - 2];
                                                        temp [3] = string [length - 3];
                                                        temp [2] = string [length - 4];
                                                        temp [1] = string [length - 5];
                                                        temp [0] = string [length - 6];
                                                        movingString(temp, 6);
                                                        free(temp);
                                        }
                                        else
                                        {
                                                        movingString(string, length);
                                        }
                                }
                        }
}
```