Time-Traveling Morse Code Puzzle

Group 61: Luca Bastone-Mohabir, Ahnaf Shahriar

02 November 2021

1 Needs Assessment

The Time-Travelling Morse Code device is designed to be an engaging multiplayer puzzle. Set in a time travelling scenario, two teammates must communicate together effectively using morse code in order to solve the puzzle and escape from the past.

1.1 Requirements

The device must be deployable as a puzzle in an escape room situation. As such, it must be challenging and require a clever solution. Furthermore, it must have clearly defined winning and losing conditions. The user must be able to interact with the device through a button to enter Morse code. It must then be able to test the user's inputs to determine whether or not the winning conditions have been satisfied. Additionally, the puzzle should be able to communicate morse code messages to the players audibly via a buzzer and visually through flashing LEDs. Finally, upon successful completion of the puzzle, the device must be able to clearly inform the players. Throughout all of this, the puzzle must remain interactive and fun for the users and it must build on the story set by the escape room.

1.2 Restraints

There are numerous restraints placed on the development of the puzzle. To begin, there are strict deadlines which must be met allowing less than a month to build a working prototype of the device. Furthermore, there is a budget constraint limiting the components which may be used in its construction. Additionally, the project must use a Nucleo 64 STM32 board which restricts the amount of computational power that can be used and must be programmed using C.

1.3 Existing Solutions

Morse code was invented in the 1830s by Samuel Morse [1]. The concept behind it is relatively simple requiring only short and long pauses to implement (represented by dots and dashes respectively when written). Consequently, it was the method of choice for long distance communication via telegraphs. The puzzle device implements morse code as a medium to deliver a puzzle to the player. Historic uses of morse code required the user to subjectively disern whether a pause should be interpreted as a dot or a dash. In contrast, the puzzle determines this objectively based on the length of the pause (i.e. if the button is held down for over a certain length of time, it is considered a dash). Furthermore, the software is able to translate the message directly into latin characters.

2 Analysis

2.1 Subsystems

Our team has identified the following subsystems which must be implemented for the device to function as intended:

- Storing Morse Code Strings
- Handling User Input
- Circuits

When selecting a solution for each subsystem, we considered the ease of implementation, memory usage, CPU efficiency, user experience, and cost where applicable.

2.2 Storing Morse Code Strings

Fundamentally, morse code messages simply consist of a combination of dots and dashes. Naturally, this is analogous to the use of binary in computing in which values are represented via combinations of zeros and ones. This leads to the first possibility for storing morse strings: (1) Using a **char** and individually manipulating bits to store a sequence of dots and dashes. For example, if we represent dashes as ones and dots as zeros, the morse code representation of C can be stored as:

$$\mathrm{C}_{\mathrm{morse}} = - \cdot - \cdot \longrightarrow 1010\ 0000$$

Alternatively, the most straightforward method for storing morse code would be (2) using an array of **char**, with each one storing either a '.' or a '-'. For example, the morse code representation of C can be stored as:

$$\mathrm{C}_{\mathrm{morse}} = - \cdot - \cdot \longrightarrow \{\text{'}-\text{'}, \text{'}\cdot\text{'}, \text{'}-\text{'}, \text{'}\cdot\text{'}, 0\}$$

Finally, the last approach is (3) converting morse input into ASCII directly at run time, and storing the string of ASCII chars. When something must be outputted as morse, it is converted back.

$$\mathrm{C}_{\mathrm{morse}} = - \cdot - \cdot \longrightarrow \text{'C'}$$

Out of the three possibilities listed, (1) and (3) are the most memory efficient as each morse character is stored in 1 byte. In contrast, (2) requires that each dot and dash is stored as a **char** in a string literal, so a morse code character would occupy 1 to 5 bytes depending on the length.

With regards to CPU efficiency, (1) and (2) are the most efficient as no processing is required. In contrast, (3) requires that each character be converted to ASCII during runtime.

Nevertheless, (1) is the most challenging to implement because the length of a morse code character is not constant. It would be impossible to determine when the character ends (i.e. is the next zero part of the character?). (2) is the trivial to implement as dots and dashes are stored directly as ASCII characters. (3) in storing the strings, however, it adds complexity when handling user input.

Overall, the amount of memory required to store any of the options is negligable compared to the total RAM on the board. The same is true for processing power. Furthermore, all would result in the same experience for the end user as this subsystem is purely internal. Thus, the deciding factor becomes ease of implementation with **(2)** being the best option for the requirements of the project.

2.3 Handling User Input

The main difficulty which must be addressed is determining when the user is finished entering their message. Solution (1) would be to follow the morse code standard of having the player end their message with a specific sequence of characters. Alternatively, (2) the message can automatically end if the user pauses for a certain length of time (e.g. 10 seconds). Finally, because the user is supposed to enter a specific sequence of characters, (3) the program can constantly compare the user's input to the expected value and then end it when either the user enters an incorrect character or the correct input is recieved.

(1) would be the most difficult option to implement as it would require checking previously inputted characters to determine whether or not the correct exit sequence was entered. In contrast (2) and (3) would be much simpler to execute as they only require checking a simple condition.

When considering solutions for user input, the experience of the player is the foremost priority. (1) would be provide the most realistic morse code experience for the user. In contrast, (2) could quickly become frustrating for users, especially those who are not adept at morse code. For example, they could pause to read to morse code chart which would unintentionally end the message forcing them to start over. (3) offers an enjoyable experience for the user, albeit with less realism.

Overall, **(2)** combines ease of implementation with a satisfactory user experience making it the ideal option for this subsystem.

2.4 Circuits

The main challenge regarding the circuitry is having a circuit that is consistent, easy to implement, and safe for usage. The two main parts of the circuitry are: Outputs and Inputs. Availability is prioritized for this project due to time and budget constraints.

For the output part of the circuitry there are two different types of solutions: The first part of the solution is choosing either (1) a two breadboard setup for each player or (2) one breadboard for each player. Setup (1) has a budget advantage as it requires 2 less breadboards and as a consequence less wiring. However setup(2) has the advantage of being more user friendly and safer due to the distance of the buttons from the live wires. But because of the cost of breadboards it is more fitting to pick solutions (1) for this project.

The Input component has two major solutions that can be implemented, (1) An Active Low button circuit or (2) an Active High button circuit. These both require the same amount of components to implement therefore the budget is the same. The user experience is not affected because they will both transmit the same input with the same accuracy. The ease of implementation for both are equal as the software detects the respective value “0” for (1) and “1” for (2). But there is a difference in terms of power consumption because solution (2) will constantly supply power to the GPIO port because it is a closed circuit at all times. But (2) only supplies power to the circuit when the button is pressed which will save power and component wear. Therefore solution (2) makes more sense

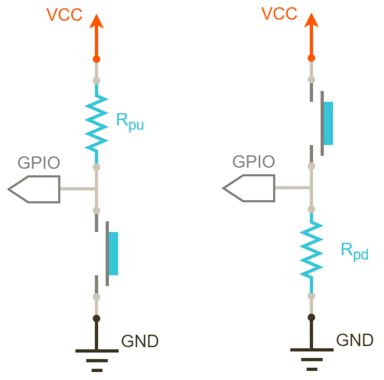


Figure 1 – (1) the left, (2) on the right

3 Implementation

3.1 Project Elements

Each of the two players will be presented with a breadboard containing a button, a buzzer, and two LEDs. All of these components will be connected to different pins on the Nucleo 64 board.
The project can then be divided into the following elements:

- Receiving User input
- Outputting morse code to the players
- Detecting winning conditions
- Circuits

3.2 Receiving User Input

When receiving user input, there are both software and hardware challenges which must be considered.

From the hardware perspective, each player is presented with a button connected to different pins on the board. They must press and hold down the button to input morse code. A buzzer sound is generated to provide audible feedback for the user.

The puzzle has two players, thus the software must be able to request input from a specific user. To achieve this, we use a function which accepts the pin attached to the appropriate button as a parameter. Each character of the user's input is compared to the required text from the puzzle so the function must take that as an argument well:

```
//returns one if the correct input is given
//otherwise, returns zero
int checkUserInput(unsigned short pin,
                   char required_input[]) {

    //loop through the required input
    for(int i = 0; required_input[i] != '\0'; ++i) {
        //get dot or dash as user input from button
        ...
        //check if the user inputted the correct char
        if(required_input[i] != input) {
            return 0;
        }
    }
    return 1;
}
```

The next challenge is determining whether the user is inputting a dot or a dash. To do so, the program must be able to tell how long the button has been held down. This can be achieved by storing the time when the button is pressed, and then comparing it to the time when it is released.

If the function returns zero, the puzzle alerts the user via the LEDs and the program loops. Once the user enters the correct input and the function returns one, the program moves on to the next component of the puzzle.

3.3 Morse Code Output

Just as each player has their own button for input, they also have corresponding LEDs to receive output from the program. Similar to receiving input, we define a function which takes a pin connected to an LED as a parameter. The message to be displayed is passed as a **char** array of dots and dashes. The length of time that the light flashes corresponds to either a dot or a dash. Spaces in the array will be treated as a pause, allowing the player to more easily determine when one character has ended and another will begin.

```
void displayMorseMessage(unsigned short pin,
                         char message[]) {
    for(int i = 0; message[i] != '\0'; ++i) {
        if(message[i] == '.') {
            //display dot
            ...
        }
        else if(message[i] == '-') {
            //display dash
            ...
        }
        else if(message[i] == ' ') {
            //pause
```

```
                ...
            }
        }
    }
```

To add further realism, the player's buzzer could also activate along with the LEDs to simulate the sound of an incoming telegraph transmission. There are two possible options for buzzers: the simplest one produces a single frequency when current is passed through it. Alternatively, the more versatile type allows the program to produce different frequencies by passing a square wave to it, however, this adds complexity to the project.

While prototyping, we wrote a helper function to convert ASCII string literals into a sequence of morse code. This is preferable to hardcoding every expected input directly as morse sequences:

```
char *char_to_morse(char message[],
                              unsigned short size) {
    //Create an array with enough space
    //to fit array of largest morse character (5 chars)
    // + a null character '\0' at the end
    morse_string = malloc(sizeof(char) * (size) * 5);

    unsigned short index = 0;

    for(unsigned short i = 0; message[i] != '\0'; ++i) {
        char c = message[i];
        if (c >= 65 && c <= 90) {
            //MORSE_LETTER: a const array mapping morse
            //characters to ASCII characters

            //Append next char to morse string
            strcat(morse_string, MORSE_LETTER[c - 65]);
        }
    }

    strcat(morse_string, '\0');
}
```

3.4 Detecting Winning Conditions

Once the players have successfully completed all of the challenges, the program must be able to inform them of their success. For the prototype, a green LED will flash rapidly to each player. In the production environment, the program could be easily modified to do something relevant to the escape room (e.g. open a box or a hidden door).

3.5 Circuits

The first challenge is space on the breadboard. In the figure below is a quick prototype of the circuit proposed for one breadboard. As observed the circuitry is very cramped because of the looping jumper wires and the length of the resistors. This combined with the small area that the STM32 has for the pins means that it looks very disorganized with wires crossing over everywhere. An improvement of this prototype in terms of space is to separate the output wiring to the left while keeping the input button to the left which would impact the safety of the user and their experience better.

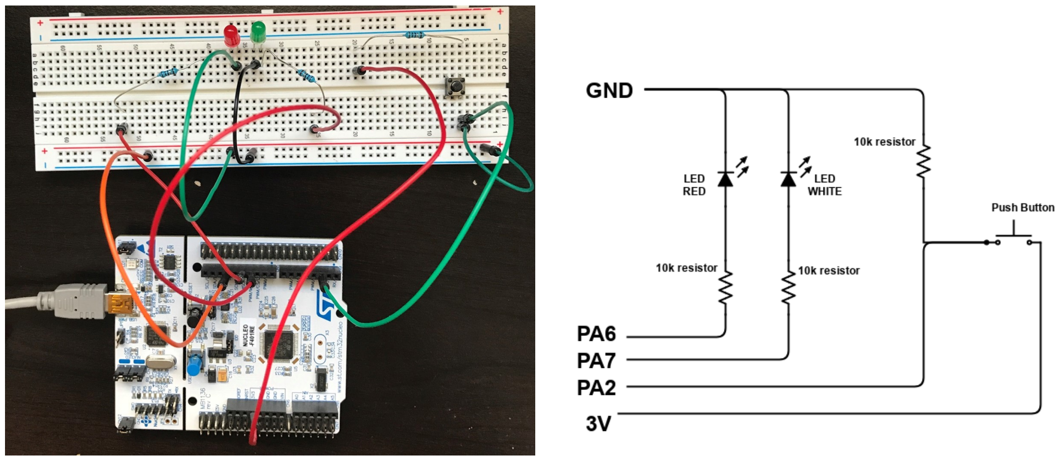


Figure 2

An additional learning from the prototype is the value of using the same ports across LEDs and buttons. This small change in hardware makes software initialization easier while also helping to keep player input and output separated i.e using player 1 circuits in GPIOA ports such as PA6 and using player 2 on GPIOC ports such as PC5.

The final challenge for the hardware components is constructing the proper circuit with the correct current rating for the given components. Such as the LEDs that were in the prototype were rated 75mA. Therefore any higher than such a current would damage and potentially explode the LED. Using Ohm's Law it is easy to calculate the required resistors in the circuit to keep it safe and extend its lifespan. Since the power was connected to 3V using V=IR, a minimum 400Ω of resistance is required. Thus the prototype used 1k resistors, which keeps it extra safe by keeping the current low for user safety.

Elements:

- 2x breadboard w/ Button for user input and LED and buzzer to display output
- 1x STM 32 Board
- Get user input from button

4 Testing Criteria

The following is the testing criteria for each individual element of the project. After each subsystem has been developed, it will be tested against the criteria in chart to ensure that it meets all of the requirements. Once the entire prototype has been created, we will test the entire game under three scenarios: a perfect game, a game with incorrect input, and one where the wrong player sends input.

Component	**Criteria**
LED	• Green LED stays off unless game is won • White LED follows button pushing
Button	• Transfers input only when pushed • Input keeps transmitting over long pushes
Buzzer	• Makes buzzer noise only when button is being pushed otherwise silent
Program Entry Point	Check for following behaviour of input conditions: • Any wrong morse input outside of expected input will restart the input • Takes input from only one player at a time
User Input	• Any wrong morse input outside of expected input will restart the input • Takes input from only one player at a time
Morse Code Output	• Sends over the input from one player to the other using buzzer sounds
Detecting Winning Condition	• Only wins after correct inputs from both players
Light Configuration	• Lights up both Green LEDs • Outputs winning morse code

5 Safety

The most outstanding issue with this design would be the open circuitry on the board. This is a risk to the player and should be mitigated. The likelihood that the board electrocutes the player on either side is fairly high because of the wiring of the resistors not being insulated. This risk should be managed by covering the resistors with insulating material such as rubber or plastic.

A further issue that was encountered during prototyping is high current in the circuitry. This was found in the first prototype of the LED as the LED and wires started to heat up due to the high current through the wires. The best solution to this is to add passive resistors in the circuitry to reduce the current flow. This would reduce the risk of damage to components of the puzzle and to the players if they were to be exposed to the circuitry.

There are also situational emergency issues with the puzzle being part of a locked room system. Losing power can cause safety issues as players would be trapped. Therefore any critical points controlled by the puzzle should have a mechanical backup to operate that point such as an emergency lever to open a door.

6 Recommendations

This project is best implemented with dedicated hardware and software teams. Most of the development can be done in parallel by the respective teams. The work should be done in segments.

6.1 Software

The software team should commence by writing a series of helper functions which will make further development of the puzzle easier. The first function to be implemented is one which converts ASCII characters into a morse code string literal (dots and dashes). This will improve code readability as correct inputs can be stored as ASCII strings rather than difficult-to-read morse.

Next, a function to output morse code via LEDs should be implemented. This function will accept a pin and morse code messages as parameters. It must then iterate through the message and activate the LED for the appropriate length of time for either a dot or dash.

The last helper function receives user input from a button. As with the output function, it accepts a pin as a parameter and an expected input as a char array. The function continuously compares the user's input to the expected one and returns 0 if the user enters something incorrect, otherwise it returns 1.

With all of the helper functions in place, the main body of the puzzle can be written. The puzzle takes place in a time traveling scenario in which player one is trapped in an undisclosed period of the past. Player two must rescue them by figuring out the exact time and location of player one. The program should start when user one inputs an "SOS" message. The program then simulates a back-and-forth conversation between the users as they exchange relevant information (i.e. player two must give information above what time period they are trapped in), once the puzzle has been solved completely, the program.

6.2 Hardware

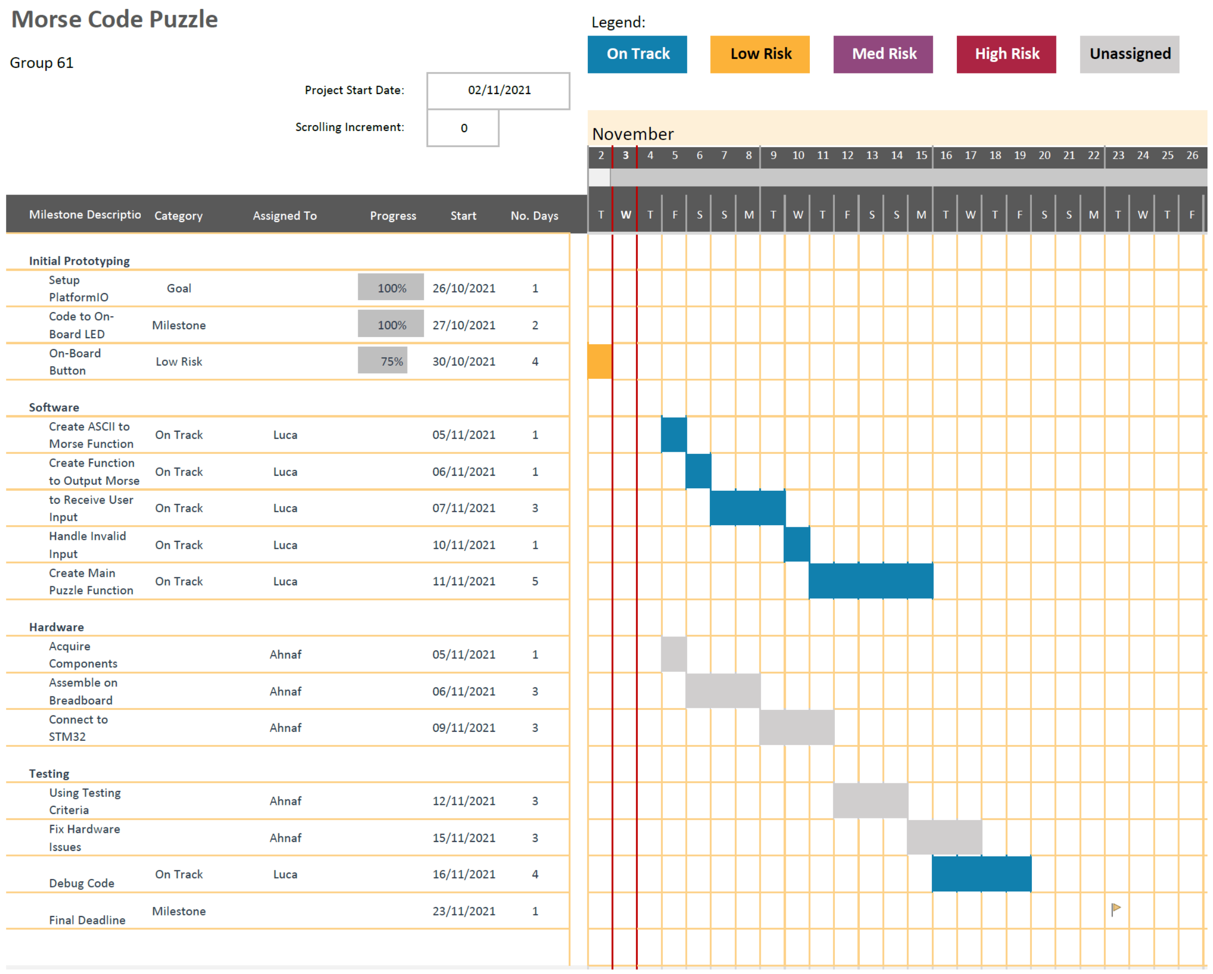
The Hardware should be assembled in 2 parts, the output and then input. Assembling the output consists of the LEDs, Buzzers with one breadboard referring to the figure 1. It is crucial that the pins match with the diagram. It is perfectly acceptable to use different GPIO ports as long as appropriate communication is given to the software team.

Then the input circuit should be assembled onto another breadboard to separate the components for safety and user experience as shown in analysis. At the end of the assembly for each part, testing should be done with the software team to validate assemblies.

6.3 Budget

Since this is a small project, only two engineers are needed for the two teams. One would have the hardware and testing responsibilities while the other engineering would be responsible for the coding. There is a 3 week time for development with a $80 budget for the components.

Component	**Cost**
STM32 FE401	$40
LED x4	$2
Wires	$15
Buzzer x2	$2
Breadboard x2	$20
Total	$80



Morse Code Puzzle

Group 61

Project Start Date: 02/11/2021

Scrolling Increment: 0

Legend: On Track, Low Risk, Med Risk, High Risk, Unassigned

November

Milestone Descriptio	Category	Assigned To	Progress	Start	No. Days
Initial Prototyping					
Setup PlatformIO	Goal		100%	26/10/2021	1
Code to On-Board LED	Milestone		100%	27/10/2021	2
On-Board Button	Low Risk		75%	30/10/2021	4
Software					
Create ASCII to Morse Function	On Track	Luca		05/11/2021	1
Create Function to Output Morse	On Track	Luca		06/11/2021	1
to Receive User Input	On Track	Luca		07/11/2021	3
Handle Invalid Input	On Track	Luca		10/11/2021	1
Create Main Puzzle Function	On Track	Luca		11/11/2021	5
Hardware					
Acquire Components		Ahnaf		05/11/2021	1
Assemble on Breadboard		Ahnaf		06/11/2021	3
Connect to STM32		Ahnaf		09/11/2021	3
Testing					
Using Testing Criteria		Ahnaf		12/11/2021	3
Fix Hardware Issues		Ahnaf		15/11/2021	3
Debug Code	On Track	Luca		16/11/2021	4
Final Deadline	Milestone			23/11/2021	1

Figure 3

7 References

[1] https://www.britannica.com/topic/Morse-Code