Technologies for Sensors and Clinical Instrumentation
Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)
Politecnico di Milano

MOUTH MORSE SYSTEM

Veronica FOSSATI
Emanuele GHILOTTI
Chiara GIOVANNINI
Lubov KOTEVA
Benedetta Lia MANDELLI
Marko MLADJENOVIC

Academic year 2020 - 2021

Contents

1. Introduction

The focus of the project is the development of a Mouth Morse (**MM**) System whose functioning is based on the interaction with the user's mouth.

The review of existing literature has allowed to identify previous uses of the thermistor for similar purposes. For example, Bandyopadhyay et al. employ the thermistor, supported by a breathing mask and a tube, as a sensing element for blow detection in an anemometer configuration [1]. This setup is of inspiration for the Mouth Morse System functioning principle: the human expiration causes an increment of the temperature that can be sensed by the thermistor and translated in the Dot-Dash characters of the Morse code, if properly temporized.

Before the development of the physical components and the conditioning circuit, it is necessary to define the target user of the Mouth Morse System. The present system is intended for subjects with severe motion pathologies and lack of coordination that prevent them from pressing a button or communicating easily in spoken language. However, they must be able to change the duration of the blow and to interface with the screen for communication purposes. Given the user conditions, every action is designed to be piloted by blowing: this aspect is implemented via software by an enhanced version of the Morse Code and associated commands which is explained in Paragraph 4.4.

For the conditioning circuit, two main solutions are available: detecting either the absolute temperature by means of two thermistors (one for the blow and the other for the environment) or the relative temperature change at the beginning of the expiration. The second option has been implemented, because it does not require any linearization and the time becomes the key factor for the translation of blowing phases into the Morse code. Indeed, the time derivative of the temperature identifies an expiration with a positive sign and disregards any other event with a null or negative sign.

Other issues are referred to the physical components that interface with the subject. First, the air outflow from the nose should not influence the sensing and the thermistor should be placed as close as possible to the mouth during communication, whilst its position can change for other activities. Furthermore, the sensing element must guarantee heat dissipation during inspiration, avoiding saturation and self-heating. In particular, the amount of humidity collected during functioning can influence the measurement.

Finally, the device is designed to be portable and wearable so that it can be used frequently in many contexts. The movement is integral with the head thanks to the lateral ties and the chin prop, as shown in Figure 1.1, for a more pleasant experience.

Figure 1.1: The face mask equipped with the thermistor during the use of the Mouth Morse System.

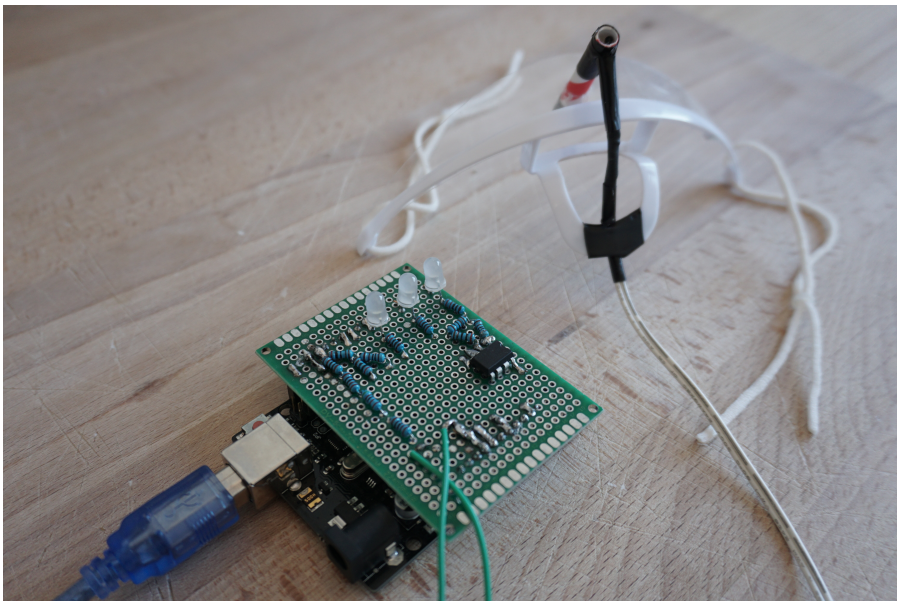

Figure 1.2: Hardware setup.

2. Hardware

The hardware of the project consists of an electronic circuit, shown in Figure 2.4: it is composed by an Arduino UNO, a NTC MF52-103 3435 thermistor, a MCP601 operational amplifier, various resistors and three LEDs. A plastic face mask supports the thermistor, which is connected via cables to the electronic circuit. The whole set can be seen in Figure 1.2.

The previous section takes into consideration many requirements that have to be satisfied in the implementation. A wearable plastic mask (shown in Figure 1.1) supports the sensing element: little effort is required in wearing it (one only needs to put the outer strings around the ears), it is a cheap item and for personal use only. A prop on the chin keeps a uniform distance from the face, reduces its invasiveness and avoids enclosing the mouth – with no formation of water vapor drops. A thin tube links the subject's airways to the thermistor and proves useful in directing the expiration towards the sensing element only if the subject has a communicative intent. The thermistor is located at the end of the tube to cool down in contact with air.

The choice of the microcontroller used for the project falls on the family of Arduino boards and in particular is chosen Arduino UNO for its greater simplicity of use in the testing phase as well as for the possibility of mounting an expansion shield on which the eletronic components could be soldered. The MF52-103 3435 NTC thermistor is a small-sized, epoxy-resin coated NTC resistor, endowed with high precision and a fast response. The resistance R_T of the thermistor varies according to the temperature T following the expression:

$$R_T = R_0 \, e^{[B(\frac{1}{T} - \frac{1}{T_0})]} \tag{2.1}$$

Where:

R_0 (10 kΩ) represents the resistance at the temperature T_0 (298 K);

B (3950 K) is the characteristic temperature of the material;

$\alpha = \frac{\frac{dR_T}{dT}}{R_T}$ is the sensitivity of the thermistor that at the temperature of 25 °C turns out to be -4.4 %

The characteristic graph of the thermistor resistance versus temperature is shown below. On the left, the whole temperature range of the thermistor is shown, whilst on the right one finds the temperature range used in the present case (Figure 2.1).

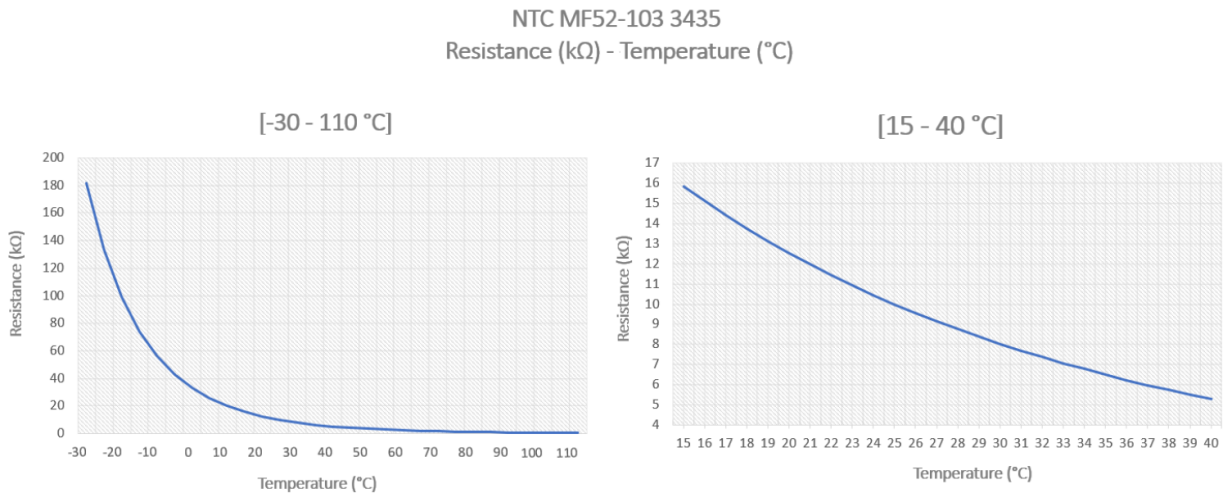

Figure 2.1: Resistance-Temperature characteristics.

The MCP601 operational amplifier from Microchip Technology Inc is chosen because of its good features and popularity (Figure 2.2). It employs an advanced CMOS technology that provides low bias current, high speed operation, high open-loop gain, and rail-to-rail output swing. This product operates with a single supply voltage that can be as low as 2.7 V, while drawing 230 µA (typical) of quiescent current per amplifier.

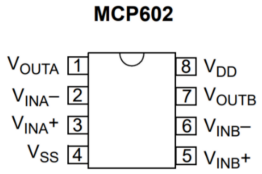

Figure 2.2: MCP602 amplifier.

The temperature sensing circuit is shown in Figure 2.3 and it is based on a resistor in series with an NTC thermistor to form a voltage divider [2]. The circuit uses an operational amplifier in a non–inverting configuration with inverting reference to offset and amplify the signal, which helps to utilize the full ADC resolution and increase measurement accuracy.

The output voltage from the conditioning circuit results:

$$V_{out} = V_{dd} \times \frac{R_1}{R_{NTC} + R_1} \times \frac{R_2 + R_3}{R_2} - \frac{R_3}{R_2} \times V_{ref} \tag{2.2}$$

For the sizing of the components these steps are followed:

1. Determine the values of R_{NTC_max} and R_{NTC_min} reachable by the thermistor and calculate the value of R_1 necessary to produce a linear output voltage.

$$R_{NTC\_min} = R_{NTC@37°C} = R_0\, e^{[B(\frac{1}{T} - \frac{1}{T_0})]} = 10000\text{k}\Omega\, e^{[3950\text{K}\,(\frac{1}{(37+273)\text{K}} - \frac{1}{(25+273)\text{K}})]} = 5.9\text{k}\Omega \tag{2.3}$$

$$R_{NTC\_max} = R_{NTC@20°C} = R_0\, e^{[B(\frac{1}{T} - \frac{1}{T_0})]} = 10000\text{k}\Omega\, e^{[3950\text{K}\,(\frac{1}{(19+273)\text{K}} - \frac{1}{(25+273)\text{K}})]} = 13.1\text{k}\Omega \tag{2.4}$$

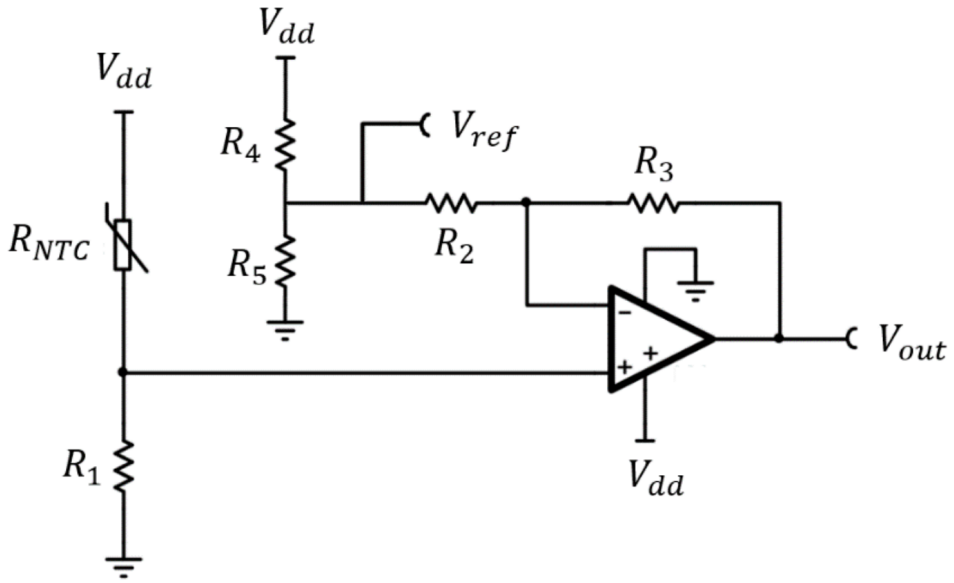

Figure 2.3: Circuit.

$$R_1 = \sqrt{R_{NTC\_min} \times R_{NTC\_max}} = \sqrt{5.9\text{k}\Omega \times 13.1\text{k}\Omega} = 8.8\text{k}\Omega \tag{2.5}$$

2. Calculate the input voltage range.

$$V_{in\_max} = V_{dd} \times \frac{R_1}{R_{NTC\_min} + R_1} = 3.3\text{V} \times \frac{8.8\text{k}\Omega}{5.9\text{k}\Omega + 8.8\text{k}\Omega} = 1.97\text{V} \tag{2.6}$$

$$V_{in\_min} = V_{dd} \times \frac{R_1}{R_{NTC\_max} + R_1} = 3.3\text{V} \times \frac{8.8\text{k}\Omega}{13.1\text{k}\Omega + 8.8\text{k}\Omega} = 1.32\text{V} \tag{2.7}$$

3. Calculate the gain required to produce the maximum output swing.

$$G = \frac{V_{out\_max} - V_{out\_min}}{V_{in\_max} - V_{in\_min}} = \frac{3.3\text{V} - 0\text{V}}{1.97\text{V} - 1.32\text{V}} = 5 \tag{2.8}$$

4. Choose R_2 and calculate R_3 to set the gain found in step 3.

$$R_2 = 1\text{k}\Omega \quad \text{(freely choosen)} \tag{2.9}$$

$$R_3 = R_2 \times (G - 1) = 1\text{k}\Omega\,(5 - 1) = 4\text{k}\Omega \tag{2.10}$$

5. Calculate the reference voltage.

$$V_{out\_max} = V_{in\_max} \times G - \frac{R_3}{R_2} \times V_{ref} \tag{2.11}$$

$$V_{ref} = \frac{-V_{out\_max} + V_{in\_max} \times G}{\frac{R_3}{R_2}} = \frac{-3.3\text{V} + 1.97\text{V} \times 5}{\frac{4\text{k}\Omega}{1\text{k}\Omega}} = 1.64\text{V} \tag{2.12}$$

6. Choose R_4 and calculate R_5 to obtain V_{ref} found in step 5.

$$R_4 = 2\text{k}\Omega \quad \text{(freely choosen)} \tag{2.13}$$

$$V_{ref} = V_{dd} \times \frac{R_5}{R_5 + R_4} \tag{2.14}$$

$$R_5 = \frac{V_{ref} \times R_4}{V_{dd} - V_{ref}} = \frac{1.64\text{V} \times 2\text{k}\Omega}{3.3\text{V} - 1.64\text{V}} = 1.97\text{k}\Omega \sim 2\text{k}\Omega \tag{2.15}$$

Once the components are sized, the circuit is assembled on a multi-hole base by means of tin soldering.
The electronic circuit, shown in Figure 2.4, is designed using Autodesk TinkerCad, a circuit editing and simulation tool. The circuit consists of the conditioning sensor part and three white LEDs in series with three 220 Ω resistors (R_6) that light up in sequence to identify the short blow (Dot), the long blow (Dash) or the very long blow (Long Dash) (Paragraph 3.1).

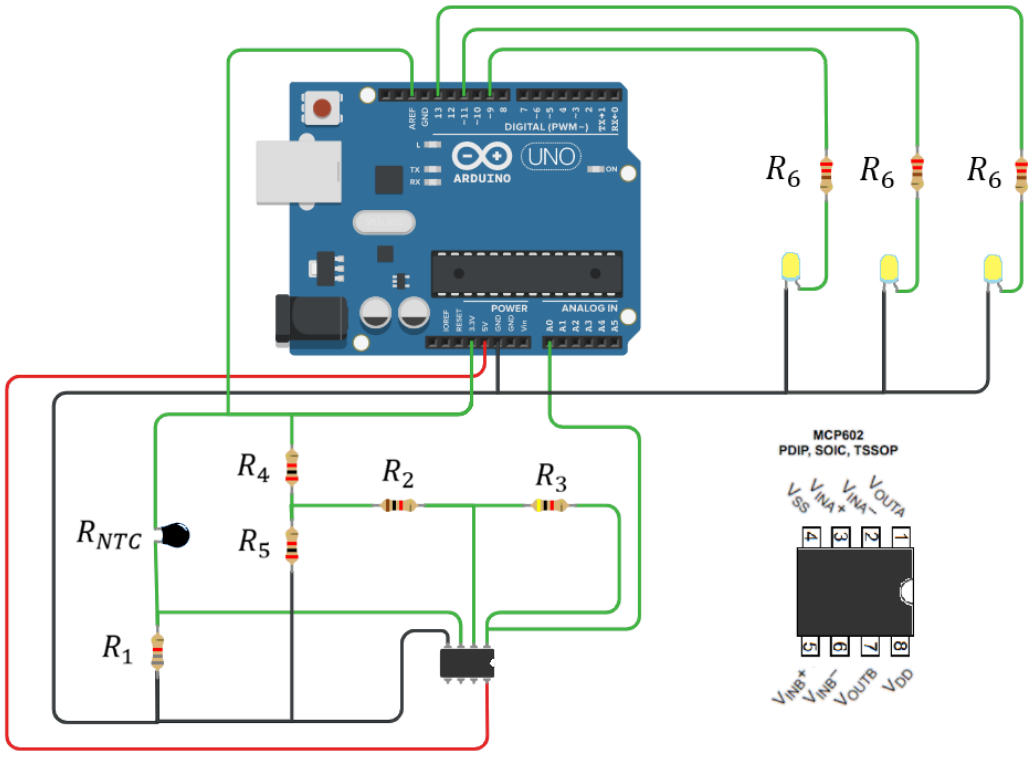

Figure 2.4: Autodesk TinkerCad circuit design.

3. Firmware

3.1 Signal sampling and Code trascription

In order to obtain an accurate detection of the user's blow on the thermistor, an Arduino code based on the variation of the temperature derivative is implemented. Since the thermistor is exposed to rapid noise generating fluctuations, it is necessary to apply a filter to the original temperature signal. The used filter is an EMA (Exponential Moving Average) first order low pass filter implemented via firmware (Figure 3.1). Its output responds more slowly to rapid temperature changes (high-frequency content) while it follows the general trend of the signal (low-frequency content).

```
filtered_output[i] = α * raw_input[i] + (1 − α) * filtered_output[i−1]
```

The (1 - α) value represents the cut-off frequency and it has been chosen experimentally (0.9 Hz). As α decreases, consequently as (1 - α) increases, the output will be less sensitive to noise.

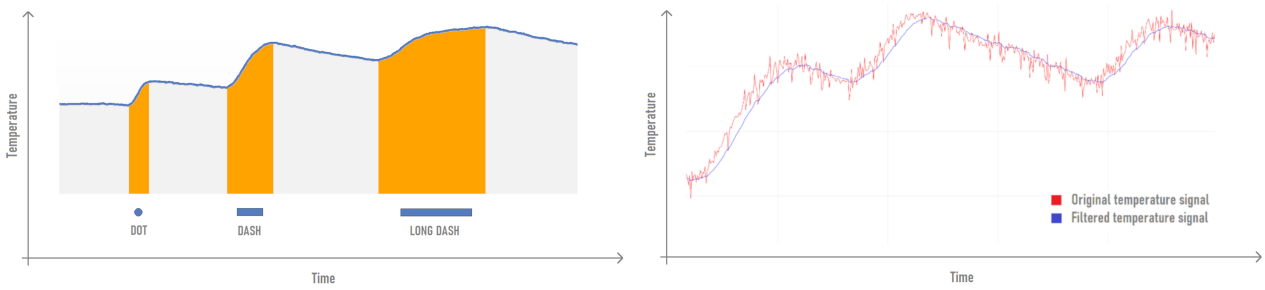

Figure 3.1: Dot, Dash and Long Dash detection (on the left). Temperature graph (on the right).

The derivative of the filtered temperature signal is then calculated every 10 ms and filtered with the same method as the temperature. By comparing the value obtained with a variable threshold, determined experimentally to avoid any noise due to involuntary blows or slight variations in room temperature, the initial instant of the blow is identified. While blowing, the amplitude of the temperature signal increases until it reaches a peak, which indicates the instant the user stops blowing.

Once the start and end times of the blow are known, the duration of the blow can be determined and compared with the predetermined time intervals corresponding to the Dot, Dash and Long Dash character.

In particular, the time interval relative to the execution of the Dot is [150, 650] ms, that relative to the Dash is [650, 1100] ms, while the one relative to the Long Dash is [1100, ∞] ms.

In order to detect the end of a command, letter or number, a 3 s timer is introduced: if the user does not execute further inputs within this time, then the string `code` containing the characters passes to translation.

3.2 Translation and Communication

In the next section of the Arduino code, the conversion of the code string into the command to be executed is carried out. To this end, an `accepted` flag is introduced to indicate whether a character-by-character match has been found. The string `code` is first compared to the dictionary `COMMANDS[SIZE_COM]` and, if no positive match emerges, to the dictionary `MORSE[SIZE]`.

In particular, `MORSE[SIZE]` contains the Morse code corresponding to the Latin alphabet letters and the digits from 0 to 9, while `COMMANDS[SIZE_COM]` is a dictionary of strings identifying special commands that allow the user to interface efficiently with the system. A third dictionary `CHARACTER[SIZE]` provides the Latin translation of the Morse code as shown in the extracts below.

```
String MORSE[SIZE] = {
  // A to I
  ".-", "-...", "-.-.", "-..", ".", "..-.", "--.", "....", "..", etc.}
String CHARACTER[SIZE] = {
  // A to I
  "A", "B", "C", "D", "E", "F", "G", "H", "I", etc.}
```

If no match is found between the string `code` and neither dictionaries `MORSE[SIZE]` and `COMMANDS[SIZE_COM]`, a `"_"` is concatenated to the string `sentence`, otherwise, the corresponding string is concatenated to `sentence`.

In order to verify the correspondence between `code` and `COMMANDS[SIZE_COM]`, a *for* cycle selects at each iteration the strings contained in the array and through an *if* condition compares them to `code`. If the match is verified, then through the use of a *switch case* code block it is possible, depending on the chosen command, to execute the corresponding action in the Processing interface (Paragraph 4.5).

- If `code` matches the space command, a further check is required. If the last element saved in the string `sentence` is a `"_"` then it is replaced with `" "`; otherwise, a `" "` is concatenated to the phrase.
- If the input `code` indicates the delete command, two scenarios are possible. If the terminal element of the string `sentence` is a `" "`, the command performs the elimination of the last 2 elements in `sentence`; otherwise, the element erased from `sentence` is only one.
- The last case verifies a `code` match with the back command identified by a single very long breath.

Afterwards, if there is no match between `code` and `COMMANDS[SIZE_COM]`, the correspondence with the strings in `MORSE[SIZE]` is checked. Again, this is implemented using a *for* loop which checks the correspondence, character-by-character, of `code` with each of the strings contained in the Morse dictionary. If a string is found to be totally equivalent to `code` then the position of that string within the array is saved and the translation of the Morse code can be found at the same position within the `CHARACTER[SIZE]` dictionary. Then the translation is extracted from `CHARACTER[SIZE]` and concatenated to `sentence` or is substituted to the last element if the last one is a `"_"`.

The communication between Arduino and Processing is structured into four strings sent by the first code to the second one. The `startBlow` and `endBlow` strings are sent to Processing in the instants of beginning and end of the execution of the breath by the user. Furthermore, a string `startGO` is passed to the Processing code which identifies both the acceptance of the last command carried out and the instant in which it is possible to start the next one. Lastly, in order to simplify the communication between Arduino and Processing, the string `character` is passed to the graphic interface containing only the last command executed and not the entire `sentence` string.

4. Software

This chapter presents the developed Graphical User Interface, its functionalities and the details of the communication from Arduino to Processing.

4.1 General introduction to MM interface

The graphic interface is entirely developed on Processing (version 3.5.4). The interface is built with the aim of being user-friendly and simple in its looks, yet incorporating the necessary basic functionalities, such as:

- Visualization of the duration of the blow with the help of the Blow Bar.
- Update about the status of the device (whether input has been correctly translated or not, when to blow or wait for the first available time slot, etc).
- Possibility to move easily among windows.
- Presence of a limited, well-defined and fixed set of commands.

The interface is developed in a state-machine fashion, such that a different value of the variable `status` activates a different window with respect to the current one or a set of options inside the window itself. Therefore, the draw function of the software consists of an *if - else if* structure:

```
void draw()
{
  if (status == LOGO) {
    gameStart();
  } else if (status == INSTRUCTIONS) {
    window_INSTRUCTIONS();
  } else if (status == INSTRUCTIONS2) {
    window_INSTRUCTIONS2();
  } else if (status == MODALITY) {
    window_MODALITY();
  } else if (status == WRITE) {
    window_WRITE();
  } else if (status == LEARN) {
    window_LEARN();
  } else if (status == LEARN_LETTERS || status == LEARN_WORDS || status ==
      LEARN_MEMORY) {
    window_GAMES(status);
  } else if (status == GETOUT) {
    window_EXIT();
  }
}
```

4.2 Windows' implementation

Each window has a dedicated function, which incorporates all the graphical and control commands, and is organized as follows:

- `gameStart()`: this is the default initial window, which introduces the user to the interface. In this window no user input is required and, while an animation of the Mouth Morse logo is shown, it allows a correct initialization of hardware and firmware. At the end of its run, it automatically sets `status = INSTRUCTIONS` to move forward.
- `window_INSTRUCTIONS()`: in this window the graphical interface and its basic elements are briefly described to make the user aware of its functionalities. The user can close the interface thanks to the **EXIT** button.
- `window_INSTRUCTIONS2()`: here the time intervals for each character are stated to the user.
- `window_MODALITY()`: the user can choose between the two main modalities, i.e. **WRITE** and **LEARN**, or return to the previous window to review the introductions.
- `window_WRITE()`: it fulfills the **WRITE** modality. For further information see Paragraph 4.6.
- `window_LEARN()`: in this overview window the user can choose among three learning games (**TYPES, WORDS** and **MEMORY**) or return to the **MODALITY** window.
- `window_GAMES(status)`: it fulfills the **LEARN** modality. For further information see Paragraph 4.7.
- `window_EXIT()`: this window is necessary to end all the processes before closing the interface.

4.3 Graphical implementation - windows' structure

Each window (except for `gameStart()`) contains a fixed number of elements to retrieve information about the user's input and the status of the system (Figure 4.1).

This fixed part is summarised in the **Control Menu**, that comprises of:

- **LED**: when the device is ready to receive a new input from the user (e.g. the start of a new letter), this element assumes a light green colour and a tick appears [3], as seen in Figure 4.1. As long as the user does not provide any input, the element remains in this state. When the user starts blowing a new letter/command and up to its correct reception by Processing, the **LED** assumes a blue-ish colour, and a loading symbol appears [4], as shown in Figure 4.3.
- **Status Bar**: it provides text messages to the user about the status of the interface. Its upper portion is synchronized with the **LED** above, showing the message *"BLOW"* when the **LED** turns green and the message *"LOADING"* when the **LED** turns blue.
- **Blow Bar**: this bar describes the length of the user's blow and is active only when a valid input is provided (i.e. when a blow duration exceeds 150ms - see Paragraph 3.1). Its level structure intuitively shows to the user whether the blow falls in the Dot/Dash/Long Dash duration range.
- **Mouth Morse logo**.

The **BACK/EXIT** button allows the user to move to the previous window or to exit the interface (when in the `window_INSTRUCTIONS()`) and it is activated by the **BACK/EXIT** command. The **CHOICE** buttons show the user all the possible alternatives in the forthcoming windows. In the depicted image (Figure 4.1),

the user can choose between two activities and will be automatically redirected to the chosen window. The instructions *"blow once"*, *"blow twice"* (which you can find in Figure 4.1) and *"blow three times"* are designed to ease the user, allowing for all possible combinations of dots and dashes corresponding to one, two or three different blows, respectively.

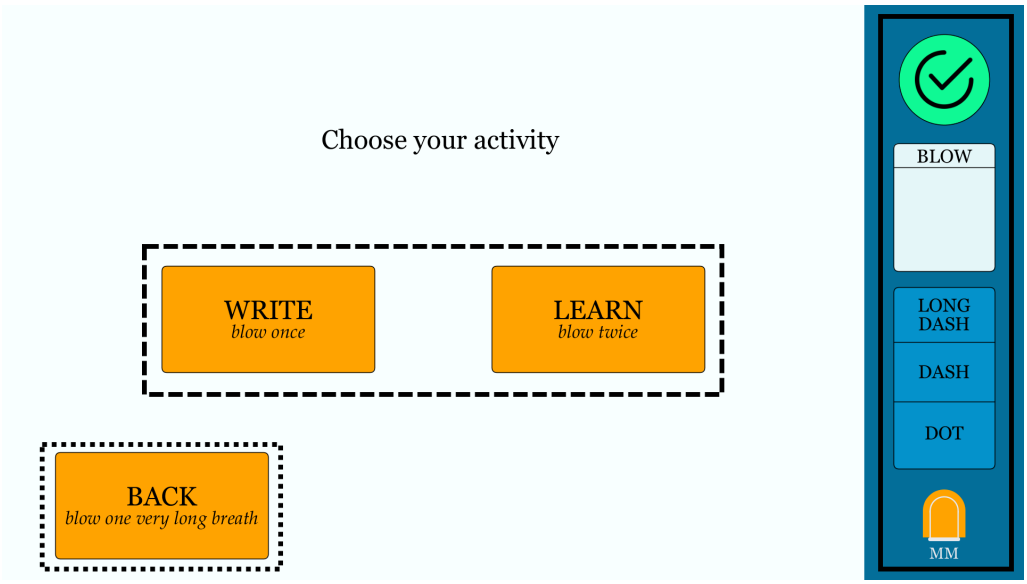

Figure 4.1: **BACK/EXIT** button (dotted line), **CHOICE** buttons (dashed line), **Control Menu** (solid line).

4.4 The "Power Morse"

The enhanced version of the Morse Code, which is the main novelty in our code, stems from the necessity to facilitate the user. The purpose is to create a short, easily repeatable, fixed sequence for each additional command needed (i.e. **BACK/EXIT**, **DELETE**, **SPACE**). Since all easily reproducible combinations of dots and dashes are already used to encode letters and numbers, the most feasible solution is to introduce a third character, the Long Dash, such that all the possible combinations of the three characters would suffice for the afore-mentioned purpose. This choice allows the command **DELETE** to be coded by a Long Dash and a Dash, while **SPACE** is coded by a Long Dash and a Dot and **BACK/EXIT** is coded by one Long Dash only.

The third character implementation is completely carried out in Arduino into the `COMMANDS[SIZE_COM]` dictionary (Paragraph 3.2), while Processing receives specific keywords for each command (for further information, see Paragraph 4.5).

4.5 Communication

The communication between firmware and software is mono-directional, since all functionalities only require information sent by Arduino. The translation of the Morse code into the corresponding Latin letter is implemented directly in the Arduino code, therefore the result must be transferred to the graphical interface through the serial port. Moreover, some functional commands are sent to Processing in order to easily manage the components of the GUI.
All the exchanged information is listed below:

- `"."` and `"-"`: they correspond to the blow of a dot or a dash.

- Numbers and Latin letters: they are sent to Processing after the translation in the Arduino code.
- `" "`: it implements the space between words in the `window_WRITE()`.
- `"_"`: it is sent by Arduino when the blown code does not correspond to any character or command.
- `"/"` or `"//"`: they indicate the **DELETE** command, the single bar for the cancellation of the last Latin character while the double bar for the cancellation of both the last character and the subsequent space.
- `"sb"` and `"eb"` (strings `startBlow` and `endBlow` sent by Arduino): they indicate the start and the end of the user's blow to manage the **Blow Bar**.
- `"back"`: it corresponds to a single long blow and it is used in every window to return to the previous one.
- `"GO"` (string `startGO` sent by Arduino): it determines the lighting of the graphic **LED**.

4.6 In-depth analysis: window WRITE

This window is the core of the entire system, since it allows the user to manage the actual communication. It shows the **BACK** button and the **Control Menu** that have been previously described. At the top there is a table which displays the Morse code corresponding to each digit and Latin letter to facilitate the user experience, while indications about the **DELETE** and **SPACE** commands are present at the bottom right. The main part of the window is the central whiteboard, in which the input message is shown while the user is communicating.

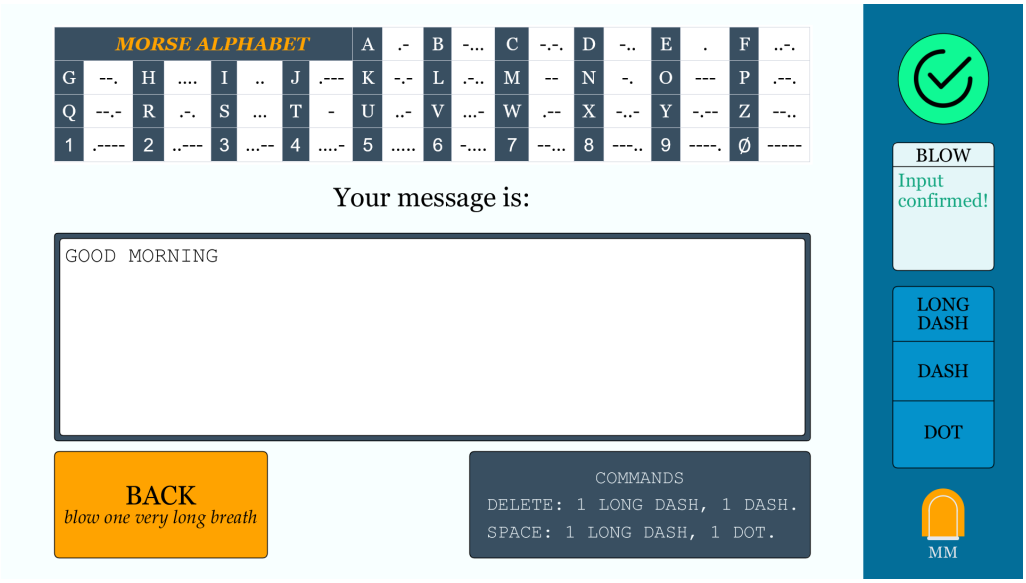

Figure 4.2: An example of a message displayed in the **WRITE** window: the last input Latin letter has been accepted and the system is ready to receive the further input, therefore the **LED** is green.

The software reconstructs the text message by receiving the different commands and characters through the serial port. When the user inputs dots and dashes corresponding to a Latin letter, the Arduino code translates them and sends the respective translation to Processing, which concatenates it to the previous text and displays in the dedicated whiteboard as shown in Figure 4.2. If a character is recognized as non-existing in the Arduino dictionaries, the error character `"_"` is displayed so that the user is provided with a visual feedback and can repeat the last character, that will directly substitute the underscore.

4.7 In-depth analysis: window GAMES

To train the user to this unusual system of communication, three games have been created. All games imply the replication of a randomly chosen type (i.e., either a Latin letter or a digit form 0 to 9) or word through the blowing rules previously explained, the user therefore has the possibility to acquire confidence with the **MM** communication method and to manage the translation times of the system in the correct way. The three possible game options are:

- `TYPES`: the user has to replicate a randomly chosen letter or number whose corresponding Morse code is shown to help complete the task.
- `WORDS`: the user has to replicate a randomly chosen word whose corresponding Morse code is shown to help complete the task.
- `MEMORY`: the user has to replicate a randomly chosen letter or number, whose corresponding Morse code is not shown in order to train the user to learn the Morse code by heart. However, if the user makes more than three mistakes, the Morse code appears on screen to help the user to complete the task.

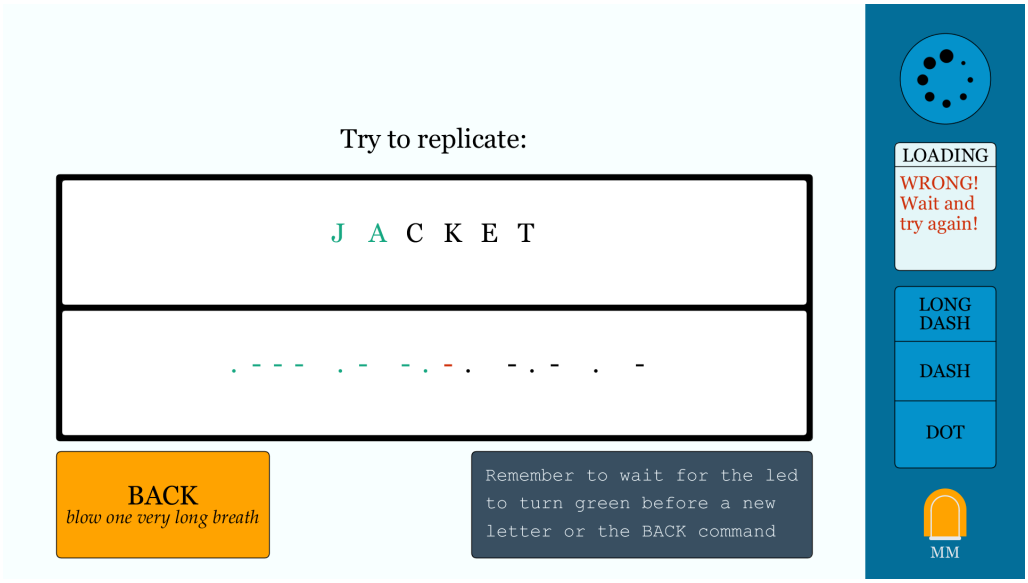

Figure 4.3: An example of the **WORDS** window after an error is recognized: the wrong message is reported in the **Status Bar**, the correct input letters and code are displayed in green while the wrong code turns red.

In each window, the **BACK** button and the **Control Menu** are present and managed as described before. The randomly chosen type or word appears in the upper section of the whiteboard, while the Morse code is shown in the lower. The user receives continuous simultaneous feedback on his performance as each Morse character is coloured in green when blown correctly and in red when mistakenly. Accordingly, different messages are reported in the **Status Bar**. The user receives a *"Right"* message after he/she manages to complete a single Latin letter, while a *"Wrong"* message is displayed if a dash is blown instead of a dot or vice versa, in this case the user will have to restart blowing from the last correct Latin letter. Furthermore, since the games are temporized to reproduce the communication time management, some messages inform the user of incorrect procedures. Indeed, after blowing successfully a single letter, the user must wait for the **LED** to turn green to proceed with the following one. If he/she is too fast, a *"Slower"* message is shown. Similarly, if he/she is too slow (see Paragraph 3.1 for the time intervals), a *"Faster"* message appears and the user has to start again from the last correctly blown letter.

5. Results

The resulting device is aimed to be a valid inexpensive alternative to communication systems that are currently on the market, such as eye tracking systems. Since it is designed for people without the possibility to move, the physical support realized through the facial mask is user-friendly and does not require external help after its placement. In the process, improvements have been introduced in order to increase the robustness and the usability and the validity of the overall implementation has been proven after several trials, indeed it has been successfully managed by different people, even with no experience of its functioning. The graphic interface's visual feedback and the detailed reported instructions make all the functionalities clear and easy to learn.

Thanks to its versatility, the system could be provided of further functionalities in order to extend its use in more complex contexts; possible further developments could imply:

- The extent of the Mouth Morse vocabulary to the whole range of ASCII characters;
- The implementation of bidirectional communication between Processing and Arduino;
- The introduction of the possibility to select a *"Beginner"* and a *"Pro"* level, which would allow adaption of the time intervals of each character to the ability of the user;
- The integration with Google Board to allow the accessibility to smartphones' settings through the blow.

A video presentation of the final interface and device during functioning can be found at this link. Different examples of the interface in use are shown below:

Figure 5.1: `window_INSTRUCTIONS()` - a welcoming message and all instructions on how to follow the **Control Menu** are displayed. The **LED** is off as the user is blowing, visible in both the **Blow Bar** and the hardware. By blowing once, the user accesses a second window which displays the 3 time intervals duration.

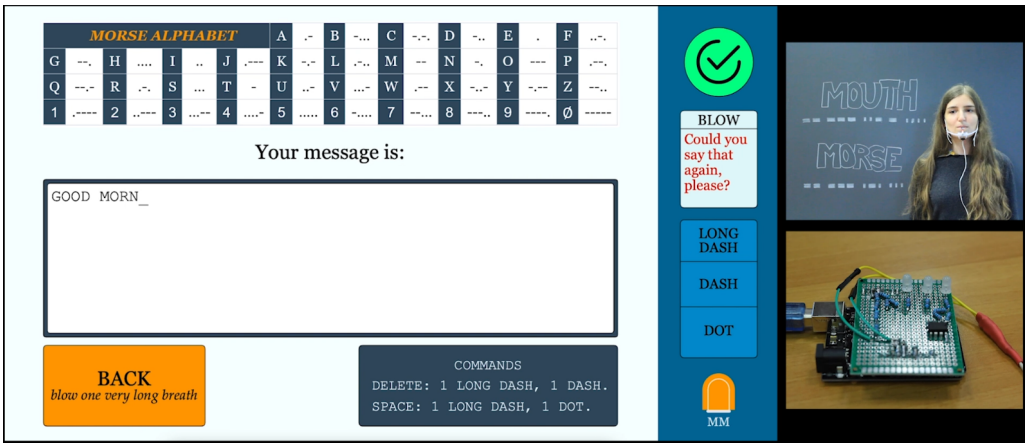

Figure 5.2: `window_WRITE()` - the underscore and the message in the **Status Bar** report an unrecognized input.

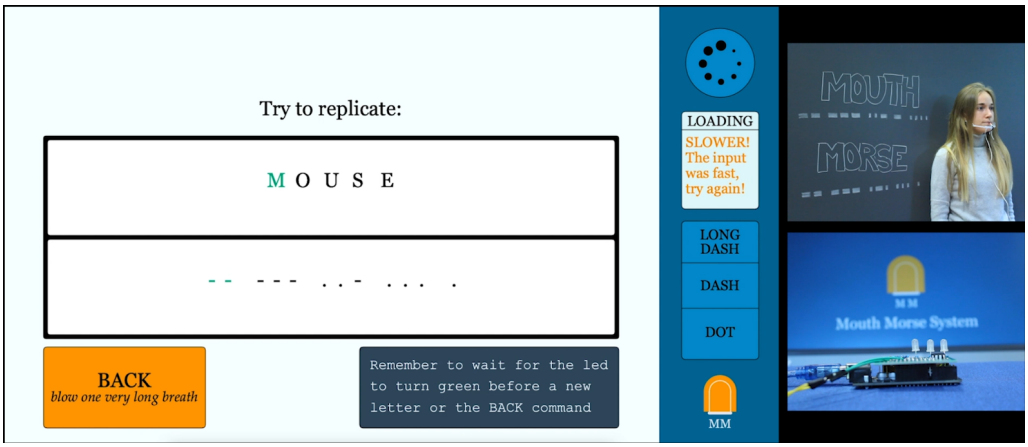

Figure 5.3: `window_GAMES(WORDS)` - After blowing correctly the first letter, if the user starts the second one too early, the message in the figure appears. On the contrary, with a delay in the letter input, the **Status Bar** displays "FASTER! The input was too slow, try again!".

Figure 5.4: `Window_GAMES(MEMORY)` - At the beginning of the game, the lower part of the blackboard is blank, without any guide or code. After three failed attempts at replicating the letter, the corresponding Morse code appears to the user. If he manages to blow it correctly, the **Status Bar** displays a message accordingly.

Bibliography

[1] S. Bandyopadhyay, A. Das, A. Mukherjee, D. Dey, B. Bhattacharyya, and S. Munshi, "A linearization scheme for thermistor-based sensing in biomedical studies," *IEEE Sensors Journal*, vol. 16, no. 3, pp. 603–609, 2015.

[2] (2018) Temperature sensing with ntc circuit. [Online]. Available: https://www.ti.com/

[3] (2013) Checked icon, designed by kiranshastry from flaticon. [Online]. Available: https://www.flaticon.com/authors/kiranshastry

[4] (2013) Loading icon, made by freepik from flaticon. [Online]. Available: https://www.freepik.com