SOP Project – Morse Code Translator

Jérémie Gremaud, Edison Kurteshi, Mathilde Voyame, Valeria Piller

Group: **404ErrorNotFound**

System-Oriented Programming 2025, BSc Course, 2nd Sem.
University of Fribourg

May 21, 2025

Main Idea of the Project

In this project, we built a communication system that sends and receives text using visible light and Morse code. Two computers can communicate with each other, connecting to a server and a webpage. The web interface lets users send messages through the Flask server to an M5Stack Atom Lite microcontroller. Each computer is equipped with an M5Stack Atom Lite and two sensors: a high-power LED called Unit FlashLight[1] and a a light sensor called Light Sensor Unit with Photo-resistance[2].

Once the user from computer A hits "send", the message is translated into Morse code and flashed using the Unit FlashLight. The Light Unit of computer B detects the signal, which is then translated and the message is reconstructed. The same process can work in the opposite direction. We created custom modules for encoding, decoding, calibration, and converted them into reusable libraries.

Architecture and Design Decisions

The architecture includes three main components:

- **Web Interface and Server**: A simple webpage communicates with a Python-based Flask server that acts as a bridge between the interface and the microcontrollers.
- **Transmitting Microcontroller**: The M5Stack Atom Lite receives a string, converts it to Morse code using a customized library, and transmits it via a FlashLight Unit by blinking.
- **Receiving Microcontroller**: Another Atom Lite equipped with a Light Unit senses the incoming flashes, processes the signal timing, and reconstructs the Morse code and the original message.

We chose to modularize the implementation by turning each core functionality into a library: Morse translation, transmission (string to light), calibration and timing capture, and decoding (light to string). This approach increased readability, reusability, and simplified testing.

Code Implementation Overview

The C code was modularized into several libraries for clarity and reuse. The core logic includes:

- **String to Light (Transmission)**: This function takes a character string, converts it to Morse code using our adapted `text_to_morse()` function, and then blinks the Flash-Light Unit accordingly. A dot is represented by 200 ms ON, a dash by 600 ms ON, with configurable gaps in between.
- **Calibration Module and data processing (Reception part 1)**: To decode light-based inputs, a system based on state-change detection was implemented. At each change (light ON/OFF), a counter resets, and the duration and value of the previous state are stored. These entries are compiled into a string like `1:200/0:200/1:600` at the end of transmission, enabling structured interpretation for Morse decoding.

[1] **Unit FlashLight**
[2] **Light Sensor Unit with Photo-resistance**

- **Binary to Morse Decoder (Reception part 2)**: This function interprets the incoming data in string format sent by the calibration and data processing function. We parse the string into Morse symbols based on duration and logical states (1 = ON and 0 = OFF) using timing thresholds with tolerances, ensuring robustness when signal readings are imperfect. We then convert it to text using `morse_to_text()` from the Morse library.
- **WiFi and Server Communication:** The initial function, `wifi_setup`, connects the controller to the specified WiFi network. Following this, the `server_setup` function uses an mDNS system to discover the server's IP address and port. Once the setup is complete, the controller communicates with the server using the `sendMessage` and `receiveMessage` functions. These functions are responsible for constructing and sending messages, as well as receiving and parsing the relevant content from incoming messages.
- **Memory Management:** We implemented a fixed-size circular buffer to store the last 16 received messages efficiently, without using dynamic memory. It uses a simple array with two indices: `head` for writing and `tail` for reading. When a new message is pushed, it's written at `head`, which advances in a loop (modulo 16). If the buffer is full, the oldest message is overwritten by also advancing `tail`. This ensures constant-time insertions and retrievals. The buffer is used to maintain a history of recent messages, replay them for debugging, and restore the system state after interruptions.

All modules were tested separately and integrated into the final workflow for reliability.

Problems encountered

The original gist from Github only handled individual characters and didn't mark word boundaries. So spaces in our text simply vanished in Morse and back. We fixed this by adding `text_to_morse()` and `morse_to_text()`, which insert "/" for spaces between words (and a single space between letters), normalize case, and use a local copy plus `strtok()` for clear, bidirectional conversion.

Each Atom Lite instead connects to the network of a PC that's running the server. As the microcontroller only manages the 2.4 GHz band, the computer must broadcast in 2.4 GHz. Of our four computers, only two were both 2.4 GHz and 5 GHz, which enabled us to connect both microcontrollers; without these two compatible computers, the ESP32 would never have been able to access the network.

Moreover, for a smooth passage of information from the calibration function to the binary to text function (light to text translation), we had to come up with a fixed string format, easy to understand and not hard to implement. This was problematic at first, since we had to make sure that both function follow the chosen format: 1:200/0:600 (state ON/OF : duration of the state and "/" as the separation between characters). At the moment of translation to morse, it had to also be compatible with the `morse.h` library, respecting the space in between the words: .... . .-.. .-.. - - - / .– - - - .-. .-.. -.. (word[*space*]/[*space*] next word).