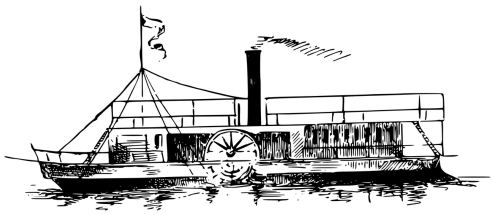
Objectif Morse

--- .·- .··· ·· ··· - --- ·· ·-· ·
-·· .·- -· ·- -·-· ·--· · ·-·
· - ·-· · - --- ··- ·-·

J. Aleksanderek
K. Zdybal

January, 2017



EX LIBRIS · KAMILA

For more projects similar to this one
visit us on GitHub: @camillejr
To contact us personally drop us a line at:
kamilazdybal@gmail.com

Contents

Chapter 1

Introduction

Welcome to the Objectif Morse project.

You will soon begin a journey through secret coded messages transmitted between powerful computers over tiny distances.

The main purpose of this document is to keep the record of the ideas that we had, of the solutions that we came up with and of the observations that we have made.

The second purpose is to serve as a tutorial for you, if you ever feel like embarking on the same adventure as we did. It is not very difficult! Following it, you will be able to (hopefully) accomplish the same mission! This journey will increase your knowledge in C++, Linux, Arduino, electronics and, undoubtedly, French language. Some basic understanding in all of these is requried as a prerequisite.

All the codes produced are not included in this document. You can download everything needed from the GitHub repository.

Reach out to section [1.3] to check if you have everything that you will need!

It's time to present the mission objectives...

1.1 Voici l'Objectif Morse

The idea for this project was born while reading Tintin comic books.

We have created the problem as follows:

You type a secret message on a stationary computer into the terminal. This message is translated into Morse alphabet and the signal of dots and dashes is

sent to the parallel port as high and low states. The parallel port's pins are connected to a small circuit with an LED diode, which blinks accordingly to the message translation. No spies should be observing the diode. This message is received by a phototransistor, situated in the close proximity of the LED. The phototransistor pass on the signal to the Arduino device. Arduino connected to a portable computer prints the received message on the Serial Monitor. The message is translated by reading the Arduino output and printed again in the terminal of the portable computer. The message is received and the war is won.

1.2 Project phases

This project is divided into two phases: **broadcasting** and **receiving**.

In general, these two phases don't require each other to get them working, however, the **receiving** phase needs a light trigger to obtain a signal. If you didn't build the **broadcasting** phase, you can use the light e.g. from a flashlight. Nevertheless, it's not recommended from the debugging point of view, unless you are an expert in broadcasting Morse signals by hand.

The two phases are separated by a $1mm$ air gap. What happens inside of the air gap shall forever remain a secret.

1.3 Equipment used

Here is the list of all the equipment that we have used.

ELECTRONICS:

1. LED blue diode
2. phototransistor
3. resistors: 330, 10 000 Ohm
4. a few cables
5. breadboard
6. parallel port plug with soldered cables
7. piece of paper (I know, that doesn't really count as electronics)

COMPUTATION:

1. Arduino Uno with a USB A-B cable
2. stationary computer with Linux ()
3. portable computer with Linux (Ubuntu)
4. of course you need some monitors and keyboards...

OPTIONAL (increases the interactivity of this project):

1. flashlight

Chapter 2

Broadcasting

`OBJECTIF_MORSE, PHASE: BROADCASTING` - The first phase of the Objectif Morse project is to translate the secret message from the regular text to the Morse code. This part introduces the message input using alphanumeric characters in the terminal, where the corresponding program is running. The message is translated by the program and broadcasted on an LED diode connected to the parallel port output pin.

2.1 Code description

The Objectif Morse project is coded in C++ language. It utilises the most powerful part of C++: object-oriented programming.

The code for broadcasting the coded message consists of 5 files plus a makefile.

The code is split into operating on the messages (`morse.cpp` and `morse.h`) and into sending output on the parallel port (`sendToPort.cpp` and `sendToPort.h`).

The most important constituents of each file are presented in the graph below.

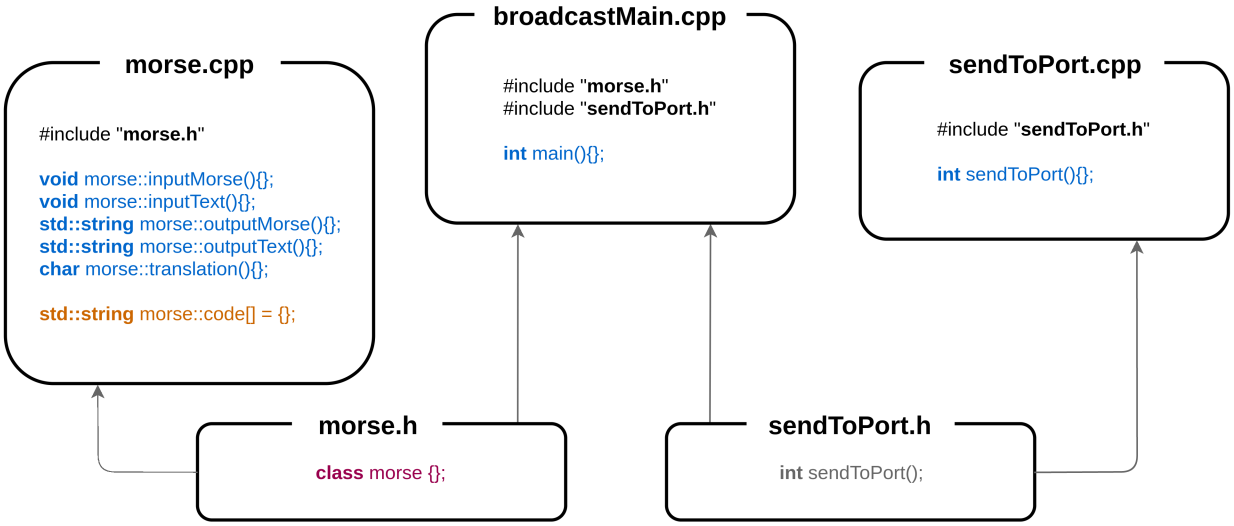


Figure 2.1: Code structure for **broadcasting** phase.

2.1.1 Class morse

We created a class called `morse` that handles all the necessary variables and functions corresponding to translating from `text > morse` and from `morse > text`.

The definition of the class is presented in the listing **??**. We will now describe the class elements.

We start with the class functions:

`morse::inputMorse()` - a public function that takes a string written in Morse alphabet as an input. Ideally, the input string should only consist of the following characters:

1. dot `"."`
2. dash `"-"`
3. space `" "`
4. slash `"/"`

If the user inputs characters other then the listed above, they will be treated as unknowns in the message.

`morse::inputText()` - a public function that takes a string written in alphanumeric as an input. The input string should only consist of:

1. letters `A-Z` (`a-z`)
2. numbers `0-9`
3. space `" "`
4. dot `"."`
5. comma `","`

If the user inputs characters other then the listed above, they will be treated as unknowns in the message.

`morse::outputMorse()` -

`morse::outputText()` -

`morse::translation()` -

Following are the class variables:

`morse::morseMessage` - is a private string that contains a message written in the Morse alphabet.

`morse::textMessage` - is a private string that contains a message written in the alphanumeric characters.

`morse::code` - is a private array of strings. A closer description of this array is given in section [2.2.2]

2.1.2 Functions

2.1.3 Sending output to the parallel port

2.1.4 Main

2.1.5 Makefile

To compile the code on your computer you can use the following makefile:

2.2 How does it work

2.2.1 Parallel port

In this section we describe how to connect to the parallel port and how to build a simple circuit with an LED diode. Notice that not every computer has got a parallel port! Usually, a modern portable computer will not be equiped with a parport, so try to get one oldschool stationary computer like we did! The parallel port is big and pink like this one:

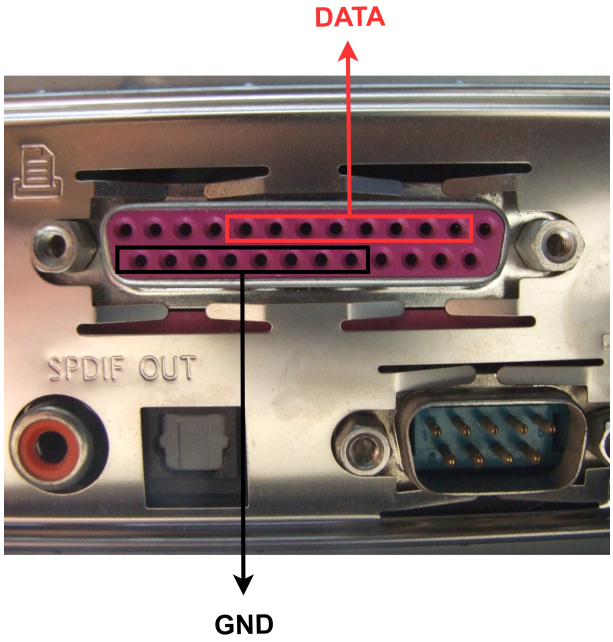


Figure 2.2: Parallel port pins: DATA and GND.

You should be mostly interested in two sets of pins on the parport - data pins (DATA) and ground pins (GND). There's eight data pins, marked in red in the picture above. They will serve as our (+) and they are the ones, who's high and low states can be controlled from the program. There's also eight ground pins, marked in black in the picture and they serve as our (-).

It doesn't really matter which ground pin you connect to. It also doesn't matter which data pin you connect to but you should adjust the number of that pin, entering new value in the place of `yourNewPinNumber` in the file `sendToPort.cpp`:

```
ioperm(base, yourNewPinNumber, 1)
```

We have connected to the zeroth data pin.

You should also be equiped with a parallel port plug. The best idea is to solder ground and data cables so that they stick firmly to the plug. Here's the picture of our plug, connected to the cables:

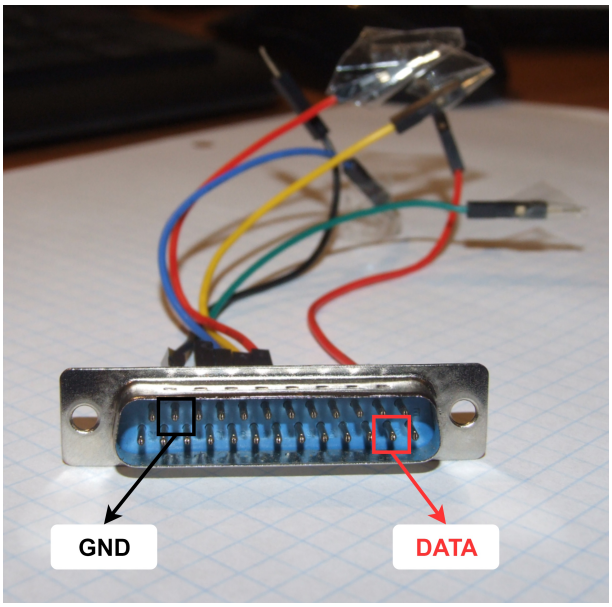


Figure 2.3: Parallel port plug.

Remember that you only need two cables - one ground and one data. You don't have to worry about other cables present in the picture above.

The final thing to do is to build the rest of the circuit connected to the plug, which is very simple! It only consist of an LED diode and a 330Ω resistor.

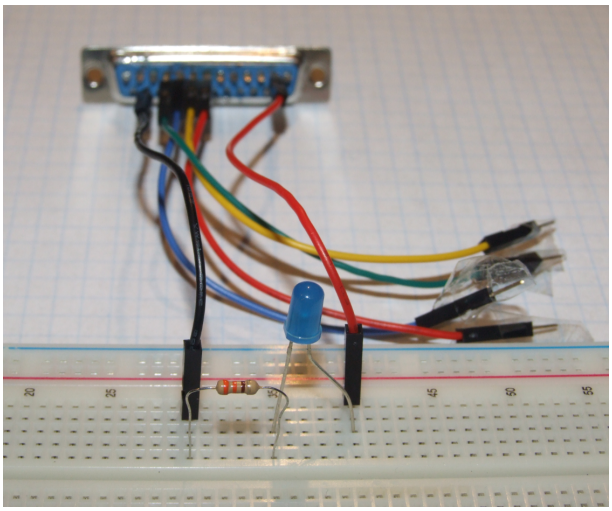


Figure 2.4: Electronic circuit for broadcasting.

2.2.2 Alphanumeric to Morse

Probably the most important part of the **broadcasting** phase is the introduction of the Morse alphabet in the code. Each letter `A-Z` (`a-z`) and each digit `0-9` has got its representative in the Morse alphabet. Its definition is present in the file `morse.cpp`. Notice that the Morse alphabet is not case sensitive and both lower-case and upper-case letters translate into the same Morse character.

We created an array of strings called `morse::code` that contains the Morse alphabet. Its graphical representation is shown below:

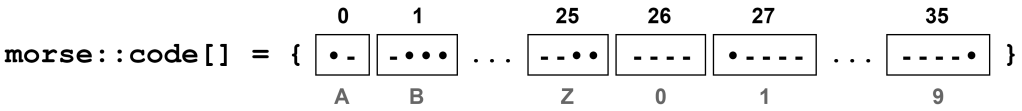


Figure 2.5: Morse code array.

Every ASCII character is assigned a number, which inside of a C++ code can be retrieved by casting the character into the integer. This is done by the following line:

```
num = (int)textMessage[n];
```

For example, for a `textMessage` character equal to `"f"` the parameter `num` will be equal to 102.

For the characters used in this project we have the following ASCII numerations:

1. upper-case letters `A-Z` : 65 - 90
2. lower-case letters `a-z` : 97 - 122
3. digits `0-9` : 48 - 57
4. space `" "` : 32
5. dot `"."` : 46
6. comma `","` : 44

We have then decided to map every legal alphanumeric ASCII character into the created array. This means that each alphanumeric character number gets the number of its position inside the `morse::code` array. This is achieved by subtracting

certain number from the ASCII numeration. The graphical representation of this mapping is presented below:

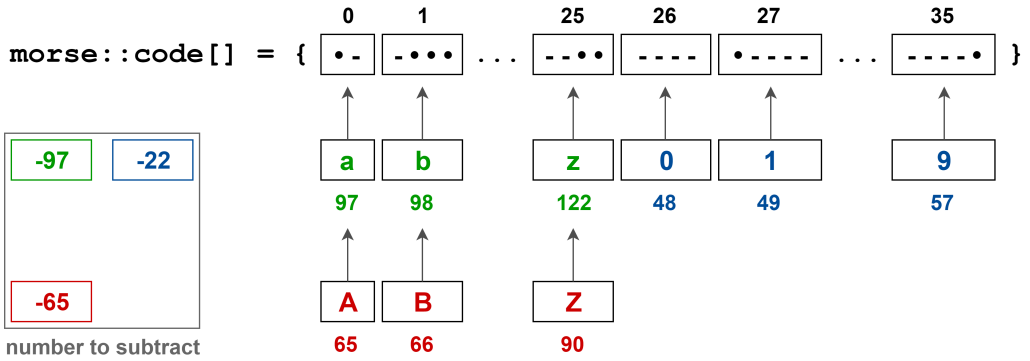


Figure 2.6: Mapping alphanumeric characters to the morse code array.

Inside of the code we have therefore:

```
if (num > 64 && num < 91) morseMessage += code[num - 65];
else if (num > 47 && num < 58) morseMessage += code[num - 22];
else if (num > 96 && num < 123) morseMessage += code[num - 97];
```

Using our previous example, the character `"f"` will result in `num` = 102, and since this number is between 96 and 123 (lower-case character), the number subtracted will be 97. This will altogether result in the position with index 5 in the `morse::code` array, which corresponds to a string "`·--·`".

This string will be added (`+=`) to the message translation variable `morseMessage`.

And so with each run of the `for` loop, the coded message will be appended by the next corresponding Morse code character.

Notice that there is also an `else` statement, which takes care of other non-alphanumeric characters which the user can type. The legal non-alphanumeric characters includes a dot, a space and a comma. Every other character is by default replaced with a space in the translation.

```
else
{
switch (num)
{
case 46:
{
morseMessage+="/";
```

```
break;
}
case 32:
{
morseMessage+=" ";
break;
}
case 44:
{
morseMessage+="/";
break;
}
default:
{
morseMessage+=" ";
break;
}
}
}
```

Chapter 3

Receiving

OBJECTIF_MORSE, PHASE: RECEIVING -

3.1 Code description

3.2 How does it work

3.2.1 Arduino part

Initials

The aim of the Arduino code is to identify the signal as Low or High and to measure the time duration of that state. One of the sample Arduino outputs is therefore:

```
L 200
H 200
L 400
H 200
L 200
H 400
```

Which has the interpretation as follows: Low state lasted 200ms, High state lasted 200ms, Low state lasted 400ms, and so on...

Variables

Variables used in the code:

`analogInPin` - specifies the analog pin that we connect to, in our case it's A0.

`sensorValue` - the value of voltage read using `analogRead()` function. This value describes the luminosity change and comes directly from the phototransistor circuit. It's a number between 0 and 1023.

`sensorMean` - the mean luminosity, assuming that the LED diode is not lit up. It is calculated in the calibration part of the code.

`sensorThres` - the upper threshold of the Low state. Any value higher then this will be considered a High state. It is calculated in the calibration part of the code.

`prevSignal` - a boolean describing the state from which the switch has just occured. It is `true` when the previous state was High and `false` when the previous state was Low.

`timeDuration` -

`timePrev` -

Calibration

The aim of the calibration is to measure the mean luminosity around the phototransistor and to use this information for further processing of the luminosity that comes from the LED diode itself.

The need to do this comes from the fact that we can send our coded messages during sunny or gloomy day or during the night, and we want Arduino to correctly interpret what part of the luminosity comes from the general brightness of the day (or night) and what part will be changes due to the LED diode broadcasting the message.

Identifying the voltage changes

Chapter 4

Remarks

- reading secret alien messages

- laser beam

Bibliography