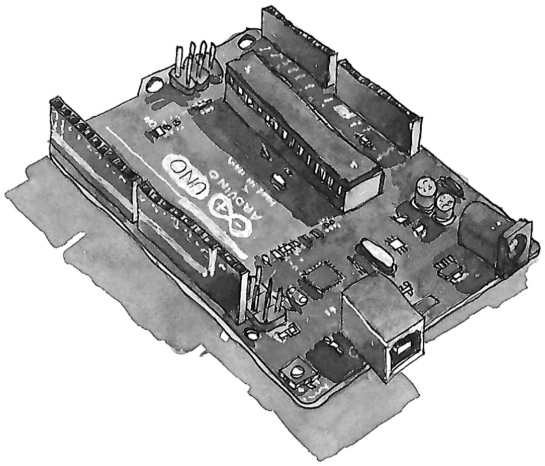
Objectif Morse

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

J. Aleksanderek
K. Zdybał

October, 2016 - November, 2017



EX LIBRIS • KAMILA

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

Objectif Morse
version 1.0

We would like to give special thanks to our friend Lama who shared her programming ideas with us when we were stuck in the project.

Thank you!

To the amazing members of the Penguin Mailing List.

« A culture of "shared knowledge" is power still needs time and passion. »

~Uncle Penguin

Contents

Chapter 1

Introduction

Welcome to the *Objectif Morse* project.

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

The first purpose of this document is to keep the record of the work that we did, of the ideas and the solutions that we came up with and of the things that we have learned.

The second and more important purpose is to serve as a tutorial for you, if you ever feel like embarking on the same adventure as we did. Following this document you should be able to accomplish the same mission and maybe even extend it with your own ideas. This journey will increase your knowledge in C++, Linux, Arduino, electronics and, undoubtedly, French language. Either some basic understanding in all of these is required as a prerequisite, or a strong will to invest time and effort to learn many new things as you go.

This project is aimed for explorers and experimenters for whom it may seem difficult at first. We believe that the spirit of "learning by doing" is the most effective way to understand the difficult.

The complete code produced is not included in this document. You can download everything needed from our GitHub repository.

Reach out to section 1.2 to check if you have everything that you need!

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

1.1 Voici l'Objectif Morse

You type a secret text 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 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 passes the signal to the Arduino device. Arduino connected to a portable computer prints the received message in Morse on the Serial Monitor. The message is translated to regular text after reading the Arduino output and printed in the terminal of the portable computer.

The message is received and the war is won.

1.1.1 Choice for the project name

The idea for this project was born while reading *Les Aventures De Tintin*, comic books written by a belgian cartoonist Hergé, where messages transmitted in the Morse code are a common form of communication.

The name *Objectif Morse* is a reference to one of our favourite books in the series *Objectif Lune* [1] (eng. *Destination Moon*), and is hence a tribute to our initial inspiration.

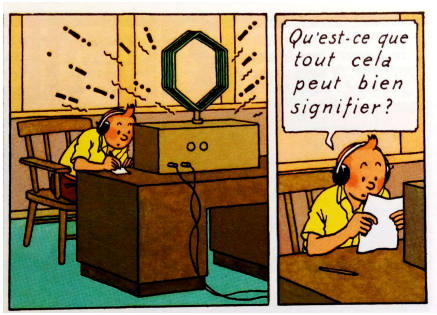


Figure 1.1: Excerpt from *Les Aventures de Tintin: Le Lotus Bleu* by Hergé [2]

1.2 Preparation

1.2.1 Project phases

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

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 for example the light 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 few centimetres air gap. What happens inside of the air gap shall remain a secret.

1.2.2 Equipment used

Electronics:

1. LED diode
2. phototransistor
3. resistors: 330 Ohm, 10 000 Ohm
4. a few cables
5. at least one breadboard but it's best to have two
6. parallel port plug with soldered cables

Computation:

1. Arduino Uno with a USB A-B cable
2. stationary computer with Linux
3. portable computer with Linux

Optional:

1. flashlight - increases the interactivity of this project

1.2.3 Code used

The *Objectif Morse* project is coded in C++ language. We've used one of its famous features - Object Oriented Programming (OOP). If you're not a fan of OOP then to make your reading more interesting we give you a full permission to silently complain about our code.

This is a scheme of how the code should be distributed over devices:

Stationary computer

`broadcastMain.cpp`
`morse.cpp`
`morse.h`
`sendToPort.cpp`
`sendToPort.h`
`makeFileB`

Portable computer

`receiveMain.cpp`
`morse.cpp`
`morse.h`
`arduinoReceive.cpp`
`arduinoReceive.h`
`makeFileR`

Arduino

`arduinoCode.ino`

In order to download our GitHub repository with all the necessary codes it's best to have `git` installed on your computers. Then type in the command line:

```
git clone https://github.com/camillejr/objectif_morse.git
```

1.3 Morse alphabet

The Morse alphabet characters are the following:

A a	·−	K k	−·−	U u	··−	0	−−−−−
B b	−···	L l	·−··	V v	···−	1	·−−−−
C c	−·−·	M m	−−	W w	·−−	2	··−−−
D d	−··	N n	−·	X x	−··−	3	···−−
E e	·	O o	−−−	Y y	−·−−	4	····−
F f	··−·	P p	·−−·	Z z	−−··	5	·····
G g	−−·	Q q	−−·−			6	−····
H h	····	R r	·−·			7	−−···
I i	··	S s	···			8	−−−··
J j	·−−−	T t	−			9	−−−−·

Figure 1.2: Morse alphabet.

The basic unit of durations of signals in the Morse alphabet is what we call in our project `dotTime`. This is the time that it takes to transmit the single dot signal. All the other durations are built from this unit and so we have:

1. the dot signal lasts `dotTime`
2. the dash signal lasts `3` × `dotTime`
3. the space between signals within one character lasts `dotTime`
4. the space between characters within one word lasts `3` × `dotTime`
5. the space between words lasts `7` × `dotTime`

There is also a slash symbol used to represent the end of a sentence. When the transmission system is capable of sending this character, it would typically last `7` × `dotTime` or more.

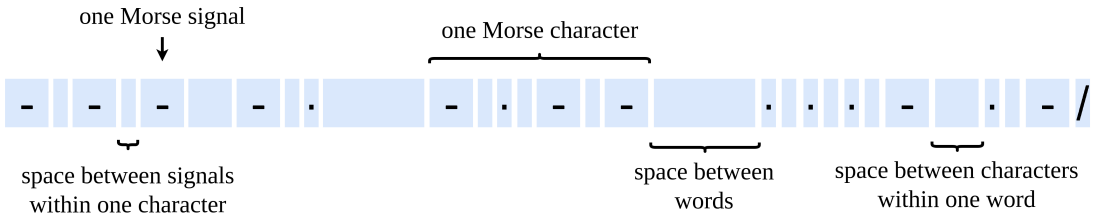


Figure 1.3: Durations in the Morse alphabet.

The Morse alphabet is scalable, which means that you can set the `dotTime` to whatever time you need, and all the remaining intervals will simply become a corresponding multiple of that unit.

1.4 Legal note

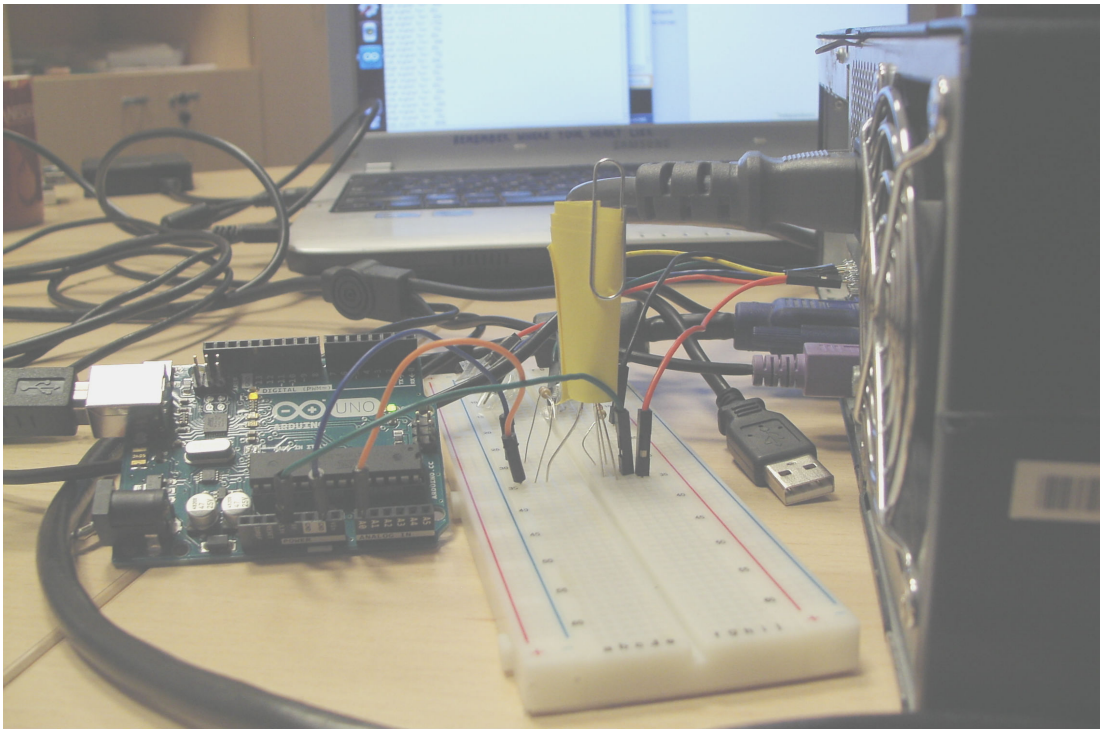
First, it is possible that things won't work when you put the cables together. Second, it is possible that the code we produced contains bugs and might cause unexpected behaviours. Third, it is possible that certain favourable conditions[1] arose when we were working on this project and it might take some effort to make them reappear again.

If you have just said: *"How awesome! I can't wait do deal with that!"*, then you're the right person in the right place. We wish you a lot of errors as you go because you're going to learn a lot from them.

[1] It is quite common among computers that the nearby presence of certain objects, or certain position with respect to certain constellations, or certain mysterious words in certain forgotten config files is causing things to work.

Chapter 2

Broadcasting



`OBJECTIF_MORSE, PHASE: BROADCASTING` - the first phase of the *Objectif Morse* project is to translate the secret message from regular text to Morse alphabet. This part comprises the message input into the terminal where the broadcasting code 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 code for broadcasting the secret message in Morse consists of 5 files plus a makefile.

The code is split into user input/output interactive part (`broadcastMain.cpp`), into functions that operate on messages (`morse.cpp` and `morse.h`) and into sending output to the parallel port (`sendToPort.cpp` and `sendToPort.h`).

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

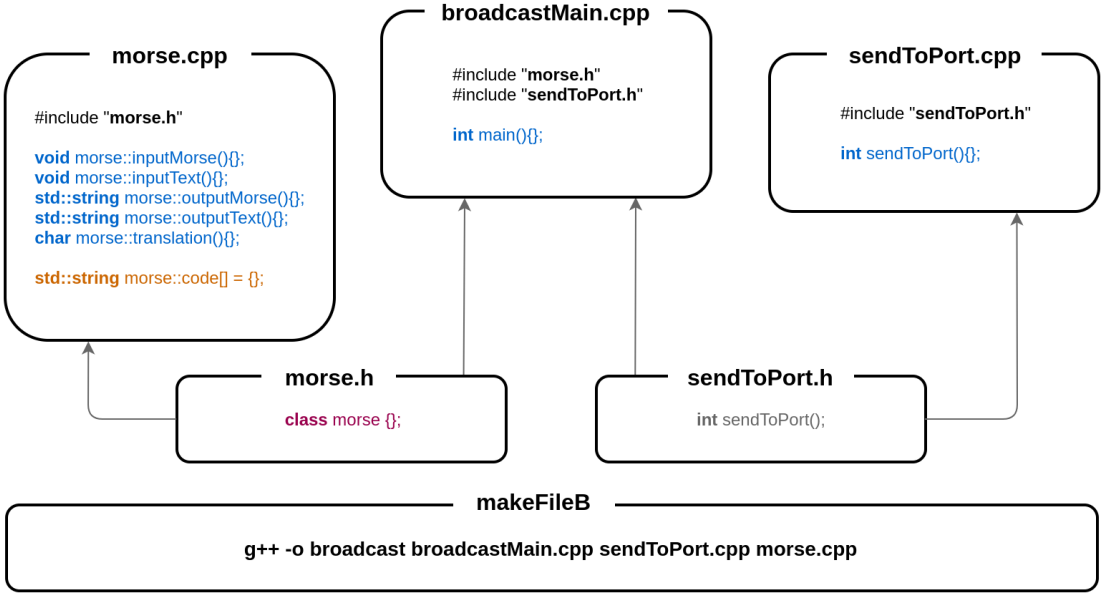


Figure 2.1: Code structure for the **broadcasting** phase.

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 below:

```
class morse
{

   public:
   void inputMorse(std::string);
   void inputText(std::string);
   std::string outputMorse();
   std::string outputText();

   private:
   std::string morseMessage;
   std::string textMessage;
   static std::string code[];
   char translation(std::string);

};
```

2.1.2 Functions and variables

In this subsection we are going to take a closer look into what the functions and variables declared inside the `morse` class do. This part is for you to get a quick overview and to help you glue things together in your head before you read the section 2.2, which describes the inner cogs much deeper.

Functions

`morse::inputMorse()` - 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()` - takes a string written in alphanumeric characters as an input. The input string should only consist of:

1. letters `A-Z` (`a-z`)

2. numbers `0-9`
3. space `" "`
4. full stop `"."`
5. comma `","`

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

`morse::outputMorse()` - function that returns the ready string consisting of message written in Morse alphabet.

`morse::outputText()` - function that returns the ready string consisting of the message written in alphanumeric characters.

`morse::translation()` - translates a series of dots and dashes from the Morse message to the ASCII sequence.

Variables

`morse::morseMessage` - contains message written in Morse alphabet.

`morse::textMessage` - contains message written in alphanumeric characters.

`morse::code` - an array of strings that create the Morse alphabet table. A closer description of this array is given in section 2.2.2.

2.1.3 Main

The "main" of the broadcasting phase is included in the file `broadcastMain.cpp`. This is where a lot of messing around can be performed, because this code is just calling our already preciously prepared pieces. If you don't like our implementation of the user interface[1] in the *Objectif Morse* project, you have a lot of freedom to play with this file.

There is one interesting behaviour that can be controlled from the `main` function - speed of the transmission, or more precisely - the duration of one dot signal (see 1.3) in the Morse alphabet. This duration in microseconds is initially set to 25000 (0.25 second) when calling `sendToPort` function:

```
1 sendToPort(secretMessage.outputMorse(), 1, 25000);
```

[1] brave words.

but can be adjusted to your needs.

2.1.4 Fire it up with makefile

To compile the code on your computer you can use the makefile `makeFileB`.

Simply teleport yourself to the place where the 5 files are placed and at that location run from the command line:

```
make -f makeFileB
```

This will result in the creation of one more file, a binary called `broadcast`.

To fire up the whole code structure (in fact, to primarily run the `main` function from `broadcastMain.cpp`) type in the command line:

```
./broadcast
```

In certain cases you may need to run the binary as `root`:

```
sudo ./broadcast
```

2.1.5 Test run

One thing that you can do right now is to ask the program to translate something for you! After the code is started you should see:

```
Select translation:
1) Text to Morse
2) Morse to text
```

Type either 1 or 2 to select an option and then input the message to translate!

As long as the electronics isn't connected to the parallel port, when you select 1, nothing more spectacular should happen.

Any errors in trying to send the message to the parallel port will result in an error message at the end of the translation:

```
Select translation:
1) Text to Morse
2) Morse to text
1
Type text message:
Hello world
You've typed:
Hello world
Morse code:
.... . .-.. .-.. ---   .-- --- .-. .-.. -..
Error sending message to parallel port.
```

The functionality to translate from Morse to text has been added mostly for fun here. When you input message directly in Morse alphabet it won't be transmitted, this is because our initial objective was to input secret message in regular text. Feel free to translate the subtitle from the front page of this document!

2.2 How does it work

This section is diving a little deeper into an explanation of the ideas behind the **broadcasting** phase. It also tells you how to set up the electronics part and hence we hope that things will get much more fun right now!

2.2.1 Parallel port connection

In this section we describe how to connect to the parallel port (parport[2]) and how to build a simple circuit with an LED diode. Notice that not every computer has got a parallel port. Usually, a portable computer will not be equipped with a parport, so try to get one old-school stationary computer like we did! The parallel port is big and pink like this one:

[2] if you wish to sound more technical.

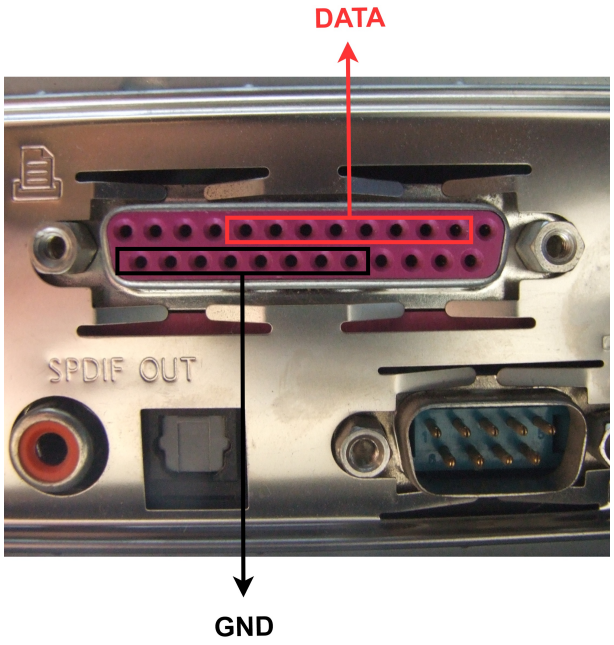


Figure 2.2: Parallel port pins: DATA and GND.

You should be mostly interested in two sets of pins here - 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 value in the place of `yourNumber` in the file `sendToPort.cpp`:

```
ioperm(base, yourNumber, 1)
```

By this value you specify up to which DATA pin you give the permission to connect to. For example, if you set `yourNumber = 3`, you will be allowed to use pins: first, second and third. Since we only needed one DATA pin to connect to, we've given ourselves permission to access just the first one, so this value is 1 by default. To read more about `ioperm` (short from input/output permission) check out this page [4].

You should also be equipped 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:

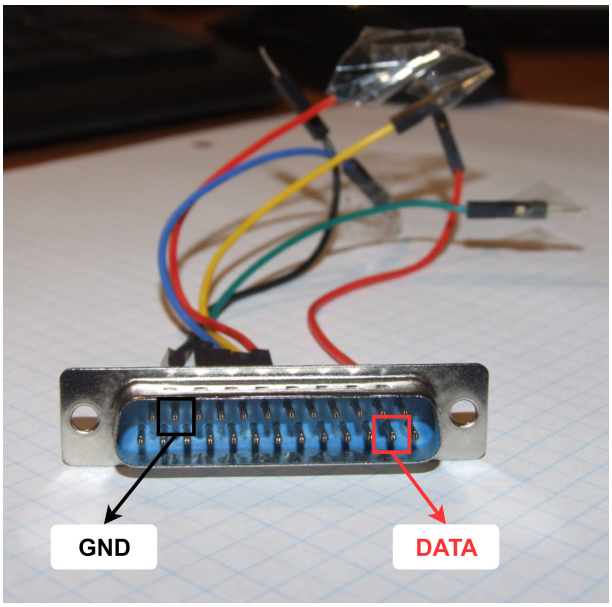


Figure 2.3: Parallel port plug.

Remember that you only need two cables - one ground and one data. You don't have to worry about other colourful cables 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.

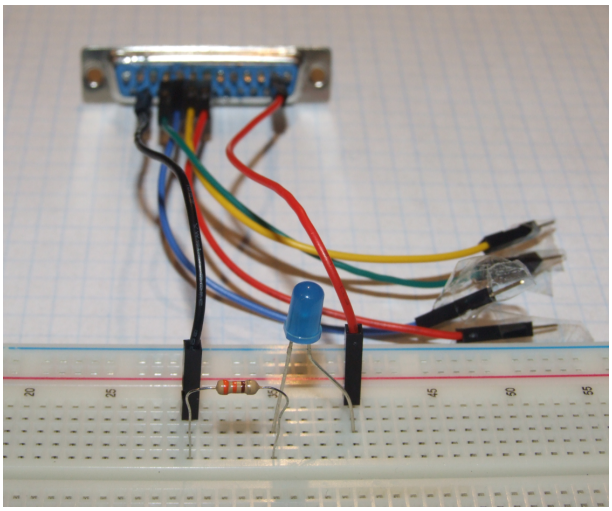


Figure 2.4: Electronic circuit for the **broadcasting** phase.

2.2.2 Alphanumeric to Morse

Probably the most important (and interesting) part of the *Objectif Morse* is the introduction of the Morse alphabet in the code. Each letter `A-Z` (`a-z`) and each digit `0-9` has got its representation in the Morse alphabet. Notice also, that the Morse alphabet is not case sensitive and both lower-case and upper-case letters translate to the same Morse character. So how can the sequence of dots and dashes be implemented in C++?

The definition of the Morse alphabet is present in the file `morse.cpp` inside the variable called `morse::code`. This is an array of strings that contains all Morse alphabet characters. Its graphical representation is shown below:

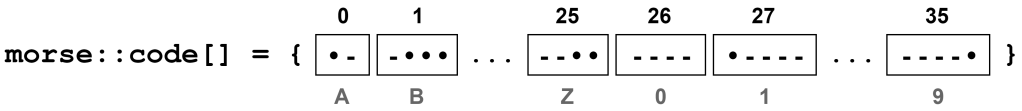


Figure 2.5: Morse code array.

Every ASCII character is assigned a number, which inside the C++ language can be retrieved by casting the character into an integer. This is done by the following line inside the function `morse::inputText()`:

```
1 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. full stop `"."` : 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 index 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:

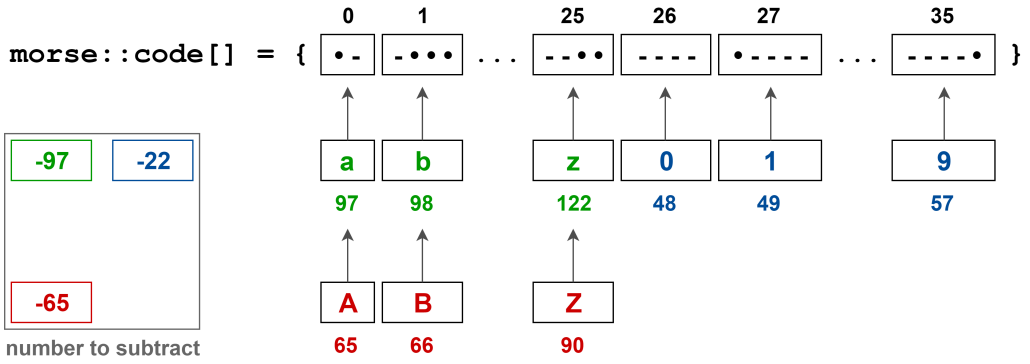


Figure 2.6: Mapping alphanumeric characters to the morse code array elements.

Inside the code we have therefore:

```
1 if (num > 64 && num < 91) morseMessage += code[num - 65];
2    else if (num > 47 && num < 58) morseMessage += code[num - 22];
3    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`.

With each run of the `for` loop in the `morse::inputText()` function, the coded message will be appended by the next 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 full stop, 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;
    }
  }
}
```

2.2.3 Morse to alphanumeric

The translation from the Morse alphabet to the regular text is taken care of by two `morse` class functions: `morse::inputMorse()` and `morse::translation()`.

The function `morse::translation()` contains a simple reverse process to what the `if()` in the function `morse::inputText()` was doing. So let's take a closer look:

```
if (!tempString.empty())
{
   for (int k = 0 ; k < 36 ; ++k)
      {
         if (tempString == code[k])
         {
            if (k > 25)
            {
               return (char)(k + 22);
            }
            else
            {
               return (char)(k + 97);
            }
            break;
         }
         if (k == 35)
         {
            return "_";
         }
      }
}
```

This `if()` statement takes one Morse character as an input. An example `tempString` might for instance be equal to "·--·". Then it runs through the `morse::code` array and checks if the character matches any of the entries inside the array. If it does, then it's going to be either a letter or a number written in the Morse code.

As previously described, all letters have indices 0-25 and all numbers have indices 26-35. Instead of subtracting a number we now have to add it to the index in the `morse::code` array to obtain the corresponding ASCII numeration. The returned value is an integer casted to a character, either:

```
return (char)(k + 22);
```

for numbers, or:

```
return (char)(k + 97);
```

for letters.

This returned value is the translation of one character from the secret message written in Morse. For example, if the `tempString` was equal to "·---", the index in the array `morse::code` is `k = 5`. Since `k` is less than 25, we know that it's a letter in ASCII and that we have to add 97 to match the ASCII numeration.

```
return (char)(5 + 97);
```

Turning the number 102 to character gives us the letter `"f"`.

The `while()` in the function `morse::inputMorse()` is making an initial preparation of a Morse character - it's building the content of the variable `tempString`. You can think of the job done by the `while()` as "slicing" the secret message to chunks consisting of single Morse characters.

This is a piece of that `while()`, without one final `case`, which we'll describe next:

```
while (n < len)
{
   tempChar = morseCode[n];
   switch (tempChar)
   {
      case ".":
      {
         tempString += ".";
         ++n;
         break;
      }
      case "-":
      {
         tempString += "-";
         ++n;
         break;
      }
      case "/":
      {
         addDot = true;
         ++n;
      }
      default:
      {
         textMessage += "*";
         break;
      }
   }
}
```

The drawing below shows how we define `tempChar`, `tempString` and `textMessage`. Each blue box would be one `tempChar` that the `while()` is currently looking at. It would contain either one Morse signal (one dot or one dash), or a space or, if the sentence has ended, the slash symbol.

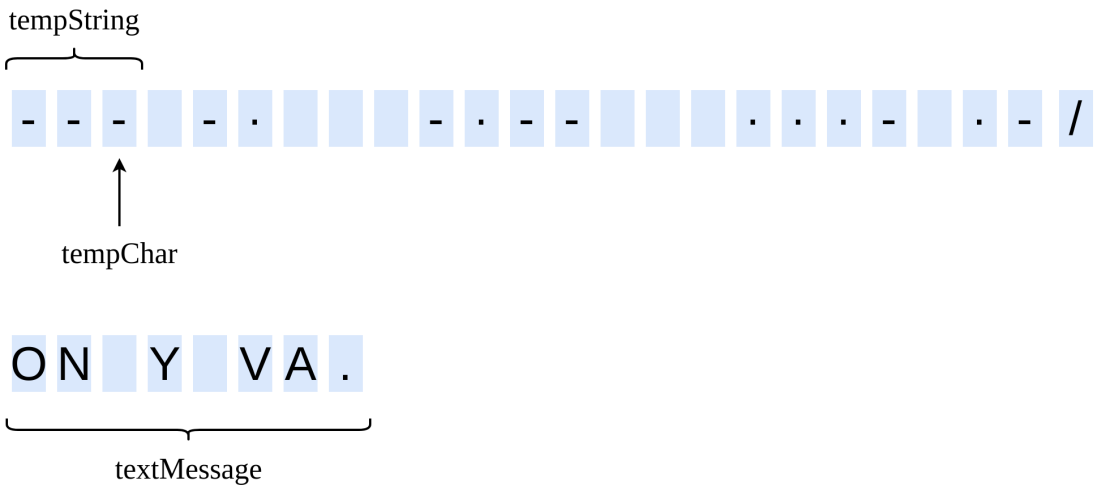


Figure 2.7: How `tempChar`, `tempString` and `textMessage` are defined.

You're maybe wondering what is this variable `addDot` all about. It simply states whether a full stop "." should be appended to the message translation. Whenever slash is present in the Morse message, it means that a sentence has ended and in the message written in regular text a full stop should be placed at the end of a sentence. So the flag `addDot` is raised to `true`, saying: *Yup! We have the end of the sentence!*

May we now bring your attention to this final `case`. The interesting thing happens when the `while()` encounters a space in the message `" "`.

```
case " ":
{
  textMessage += translation(tempString);

  tempString.clear();

  if (addDot == true)
  {
    textMessage += ". ";
    addDot = false;
  }

  int spaceCount = 0;

  while (morseCode[n] == " " && n < len)
  {
    ++spaceCount;
    ++n;
  }

```

```
21    if (spaceCount < 3)
22    {
23      break;
24    }
25
26    else
27    {
28      textMessage += " ";
29      break;
30    }
31  }
```

First of all, this means that one full Morse character has finished and it can be translated, hence the first thing to do is:

```
1  textMessage += translation(tempString);
```

This line is saying: *Hey! That's the end of a character! Dear function* `morse::translation()`*, would you please translate it for me?* And then the result of the translation gets appended to the variable `textMessage`.

Next, we have to find out what is the reason for why the character has ended. Let's look at the message as the user might input it into the terminal:

```
--- -.   -.--   ...- .-/
```

So the reason might be twofold: either it's a space between Morse characters within the same word, or it's a space which separates two words. This final `case` is counting how many spaces are there after the first encountered, and then it's classifying the reason. We have given it some flexibility as well, so you can see that when the total number of clustered spaces is less than or equal to three, we're doing nothing and just treating this as a space between Morse characters. If it's any larger than 3 we're interpreting that as a space between words and in the message translation to regular text we have to add a space as well.

```
on y va.
```

2.2.4 Sending the output to the parallel port

The code responsible for sending the message as high and low states to the parallel port is `sendToPort.cpp`.

There is only one function defined in this file and it is called `sendToPort()`. It takes as inputs two variables: the message to be broadcasted and the duration `dotTime` which defines the speed of the transmission.

But before we jump into this function's body, let's take a closer look at the declarations section. You can see the mysterious line here:

```
1 #define base 0x378
```

The hex number `0x378` defines a register address in the memory. This memory area is 8-bits long and each of its bits is directly linked to each of the 8 DATA pins on the parport. The low or high state at any bit of this area (represented by either 0 or 1) is going to be the same at the parport pin linked to this bit.

We can then tell the bit to turn its value to 1 and the state of the corresponding DATA pin will become high - meaning that if an LED diode was connected to this pin it would light up.

So that's how we're going to transmit the message to the parallel port - by simply writing the correct values to this special memory place.

If you're curious about the `#define` instruction we encourage you to read more about macros[3] in C++. All that is important to know for now is that this line is saying: whenever you see `"base"` written in the code, replace it with the value `0x378`. It's much like an alias and a somewhat more fancy way of stating that `base = 0x378`.

Now let's come back to the `sendToPort()` function and see what's coded inside!

First, you'll see the `if()` statement with the `ioperm()` function that you've already heard about in the subsection 2.2.1. This is just a check in case something goes wrong with obtaining permissions to access the parallel port.

Then comes the `for()` loop which iterates over every character in the secret message `stringToSend` and then classifies what it is using `switch()`.

[3] and you will most likely read that they are best to be avoided!

```
switch (stringChar)
{
  case ".":
  {
    outb(1, base);
    usleep(dotTime);
    outb(0, base);
    usleep(dotTime);
    break;
  }
  case "-":
  {
    outb(1, base);
    usleep(3*dotTime);
    outb(0, base);
    usleep(dotTime);
    break;
  }
  case " ":
  {
    outb(0, base);
    usleep(2*dotTime);
    if (stringToSend[i+1] == " ")
    {
      ++i;
      if (stringToSend[i+1] == " ")
      {
        ++i;
        usleep(4*dotTime);
      }
    }
    break;
  }
  case "/":
  {
    usleep(6*dotTime);
    break;
  }
}
```

There are two helpful functions that we're using. The first one is the function `outb()`, used here to write low (0) or high (1) state into the register address `0x378`. The second one, function `usleep()`, is simply telling the program to "*wait and do nothing*" for a period of time specified as its argument; we're going to call it "sleep".

Analysing the first two cases you can probably see right away what is happening: we write (1) to the register, which lights up the diode and then we sleep for a certain period of time which corresponds to the Morse signal that we're broadcasting, e.g. for `dotTime` when the signal is a dot or for `3` $\times$ `dotTime` when the signal is a dash. While we sleep the diode is turned on all that time. After that

we turn the diode off by writing a low state (0) into the register.

Notice that at the end of broadcasting a dot or a dash there is always one more sleep in the "off" state for the duration of `dotTime`. This is done in order to transmit the space between Morse signals - which lasts one `dotTime`.

Again the most interesting thing happens when the current character `stringChar` is a space `" "`. First of all, this means that one full Morse character has ended. And this would mean that it is either a space between characters within one word or a space between words. In any case, we can already sleep for another `2` $\times$ `dotTime`.[4]

Next, we check if there are still more spaces after the first encountered. If there is a space still ahead, we simply move on to the next character in the `stringChar` and we check again (in the second `if()` statement) if there is one more space ahead. This already means that there are three consecutive spaces which signifies the space between words and we have to sleep for another `4` $\times$ `dotTime`[5]

Finally, in the last `case`, where the character encountered is `"/"` we simply sleep for `6` $\times$ `dotTime`[6] and this would be the end of a sentence.

Uffff. We realize that keeping track of all these durations can be a real pain in the neck! This was a source of a lot of mistakes in our initial versions of the code. For example we forgot that we were already sleeping one `dotTime` after every Morse signal and at first we made the code sleep for too long after every Morse character. So don't feel discouraged if you didn't follow that explanation. Take as much time as you need to digest what is happening here.

Function callout

An example of the function callout could be the one from `broadcastMain.cpp`:

```
1 sendToPort(secretMessage.outputMorse(), 25000);
```

where we give as an input the message to broadcast: `secretMessage.outputMorse()` and the time duration `dotTime = 25000` microseconds.

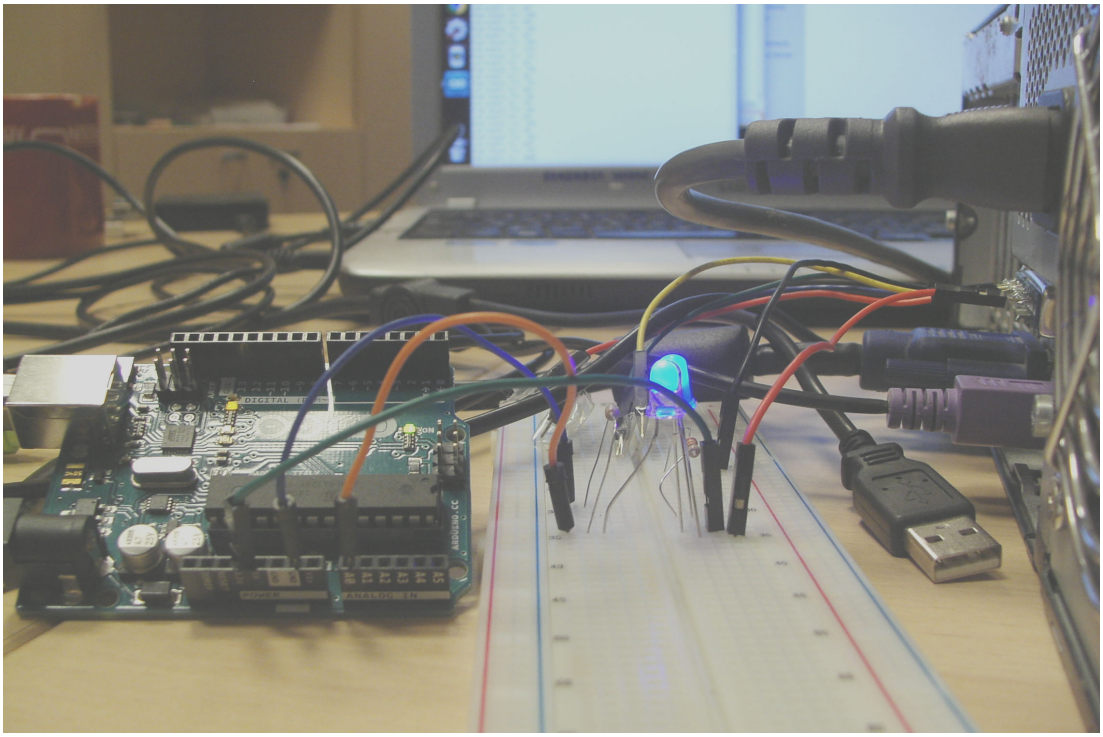
[4]observe that we already sleep one `dotTime` after every Morse signal, so the additional `2` $\times$ `dotTime` will add up to sleeping `3` $\times$ `dotTime` if this is the end of the Morse character.

[5]so that altogether we have slept for `7` $\times$ `dotTime` here.

[6]again, altogether we have slept for `7` $\times$ `dotTime` here.

Chapter 3

Receiving



`OBJECTIF_MORSE, PHASE: RECEIVING` - the second phase of the *Objectif Morse* project is to capture the secret message coming from far away in Morse alphabet and decode it to back regular text. This part begins where the phototransistor connected to Arduino receives the light trigger and Arduino passes in on to the receiving code running on a different machine for further translation and output.

3.1 Code description

The code for receiving phase consists of 5 files plus a makefile.

The broadcasting and receiving phase share the same `morse.cpp` file responsible for translation of messages. Additionally, there is a set of functions that parse the Arduino output (`arduinoReceive.cpp`, `ArduinoReceive.h`) and the output part (`receiveMain.cpp`, `receiveMain.h`) responsible for printing the result of the message translation in the command line.

The code structure is presented in the graph below:

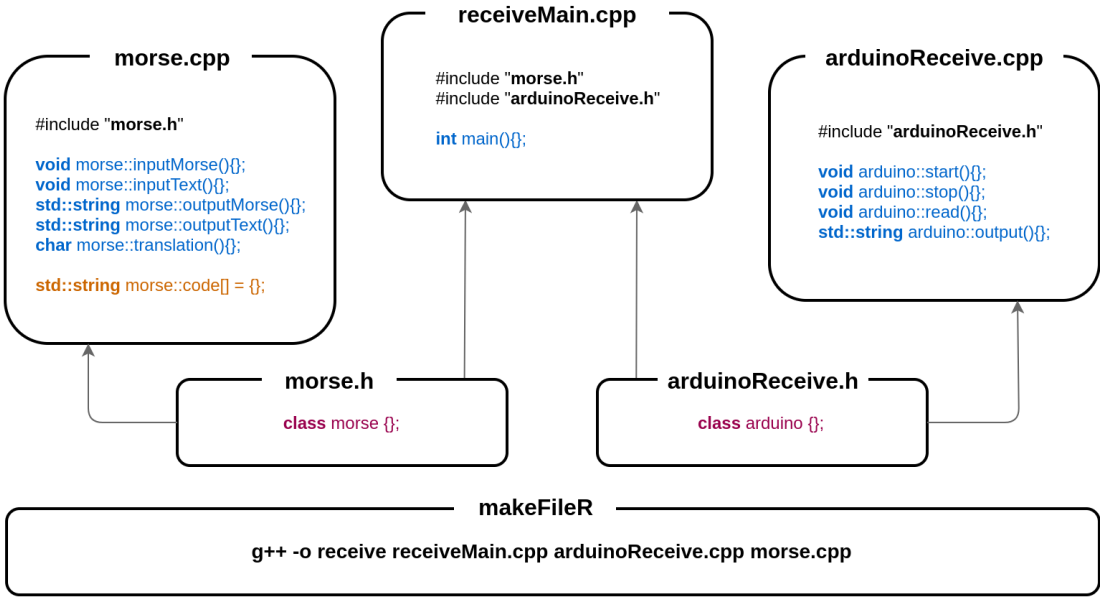


Figure 3.1: Code structure for the **receiving** phase.

3.1.1 Class `arduino`

For handling the output received from Arduino we created a class called `arduino`:

```
1  class arduino
2  {
3
4     public:
5     void start();
6     void stop();
7     void read();
8     std::string output();
9
10    private:
11    FILE* arduinoFile;
12    std::vector<int> durations;
13
14 };
```

3.1.2 Functions and variables

Functions

`arduino::start()` - function that opens the Arduino output file for reading.

`arduino::stop()` - function that closes the Arduino output file.

`arduino::read()` - function that reads the output from the Arduino output file.

`arduino::output()` - function that translates the Arduino output to a string written in the Morse code.

Variables

`arduino::arduinoFile` - contains a pointer to the Arduino output file.

`arduino::durations` - a vector of integers storing a stream of Morse signals.

3.1.3 Main

The "main" of the **receiving** phase is included in the file `receiveMain.cpp`.

3.1.4 Fire it up with makefile

The code is compiled in an analogous way to the **broadcasting** phase. Once you have all 5 files in one place run from the command line:

```
make -f makeFileR
```

A new binary file called `receive` will be produced.

To run the **receiving** phase type in the command line:

```
./receive
```

If this doesn't work, you might need to run the binary as `root`:

```
sudo ./receive
```

And the code should be up and running!

3.1.5 Test run

Whether you've already built the **broadcasting** phase or not, you can run the binary `receive` and see for yourself what the *Objectif Morse* is capable of!

When the code is started you will see the modest:

```
Input time:
```

You should type the time in seconds during which the setup will "listen" to the incomming message.

If you have a flashlight at hand you may want to play a little with broadcasting Morse signals by hand. Just type for example 30 seconds and hit Enter:

```
Input time:
30
```

3.2 How does it work

3.2.1 Arduino connection

The picture below describes how to build the electronic circuit for the **receiving** phase. It consists of a phototransistor and a 10 000Ω resistor. **Always double-check the circuit that gets connected to Arduino** because the mistakes may sometimes be sad for your little Italian device. It's a good habit to first upload the code to Arduino (only connecting it to a USB A-B cable) then disconnect the Arduino from the USB and assemble the circuit in a power-off mode.

We use the analog pin A0, as well as the 5V pin to serve as our (+) and the GND pin to serve as (-).

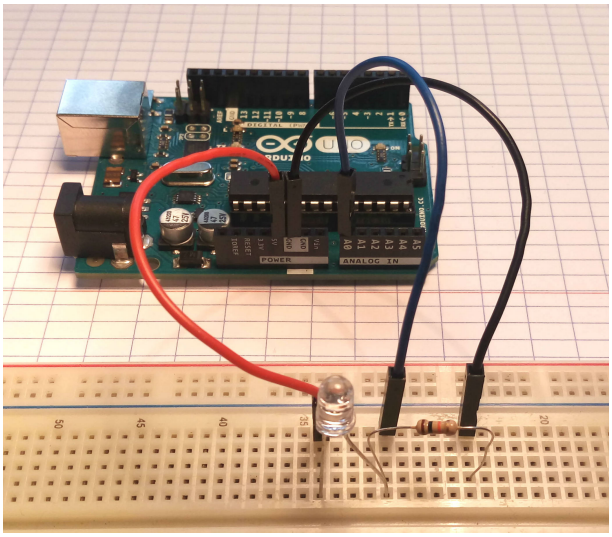


Figure 3.2: Electronic circuit for the **receiving** phase.

3.2.2 Arduino code

In this part we describe the contents of the Arduino code `arduinoCode.ino`. This code is written in C language and should be uploaded to your Arduino device. We recommend downloading Arduino IDE for Linux - a neat little program in which you can easily verify and upload Arduino codes. It also comes with a library of examples that could be useful in your future projects.

Arduino's job

The aim of the Arduino code is to identify the signal as Low (`L`) or High (`H`) and to measure the time duration in *ms* of that state.

One of the sample Arduino outputs can be:

```
L 200
H 200
L 600
```

Which has the interpretation as follows: Low state lasted 200 *ms*, then High state lasted 200 *ms*, then Low state lasted 600 *ms*, 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 as seen directly by the phototransistor. 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 lower threshold of the High state. Any value higher then this will be considered as a High state. It is calculated in the calibration part of the code.

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

`timeDuration` - the total time of the last Low or High state.

`timePrev` - the time in *ms* at which the change from Low to High or from High to Low occured, measured from the beginning of the program operation.

In the graph below we present in a closer detail how the Low and High state is interpreted by the Arduino code. Anything that is higher then `sensorThres` is interpreted as a High state. Anything that is 10 from `sensorMean` (either way) is interpreted as a Low state. There are two gaps where the signal is not interpreted

as either High or Low. The explanation for why we've made it that way is present under **Blackout zone explanation** at the end of this subsection.

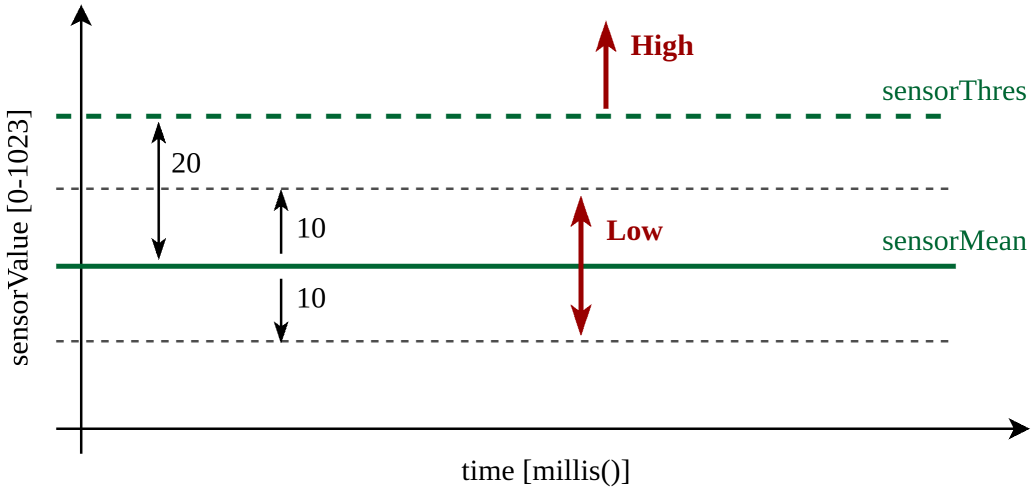


Figure 3.3: Interpretation of Low and High states using defined variables.

Calibration

The aim of the calibration is to measure the mean luminosity around the photo-transistor and to use this information for further processing of the luminosity that comes from the LED diode itself. This mean luminosity will be present everytime we have a Low state in our signal - interpreted as "darkness".

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 the Arduino to correctly interpret what part of the luminosity comes from the general brightness of the surroundings and what part are the changes due to the LED diode.

The aim of the calibration is to obtain the value `sensorMean`. This is performed by reading 10 values of `sensorValue` in 100 *ms* time steps and calculating their arithmetic average.

Identifying the voltage changes

Inside the `loop()` function, the `sensorValue` is read at the beginning and then if certain conditions are met, we have either signal changing from Low to High or changing from High to Low.

The signal is changing from Low to High when the voltage read is greater than

the `sensorThres` and when the previous signal was Low (when the `prevSignal` is set to false). This condition is comprised in the following `if()` statement:

```
1 if (sensorValue > sensorThres && !prevSignal)
```

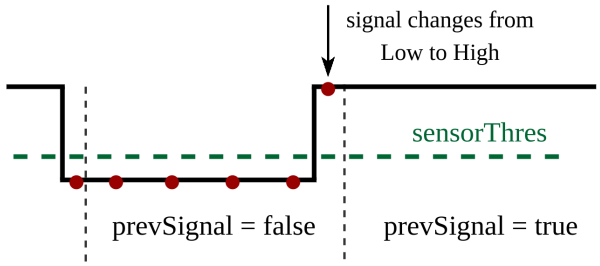


Figure 3.4: Signal changing from Low to High. (Red dots represent the `sensorValue` measurements.)

The signal is changing from High to Low when the voltage read is within 10 from the `sensorMean` value and when the previous signal was High (when the `prevSignal` is set to true). This condition is comprised in the following `if()` statement:

```
1 if (sensorValue - sensorMean < 10 && prevSignal)
```

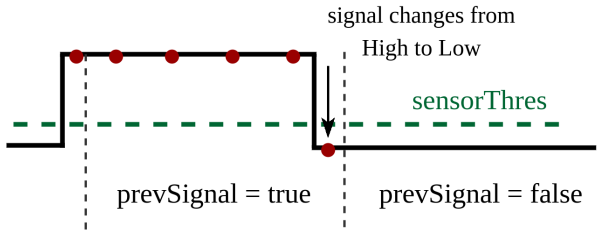


Figure 3.5: Signal changing from High to Low. (Red dots represent the `sensorValue` measurements.)

Measuring the time durations

Each time we enter one of the two `if()` statements, the time of the last finished state (High or Low) must be identified, measured and printed on the Serial Monitor.

Additionally, a flag `prevSignal` is always changed to describe the currently entered state.

For the purpose of measuring the time durations we use the built-in Arduino function `millis()`. This function measures the time in *ms* that has passed from the beginning of the operation of the program. It can therefore be viewed as an absolute time. It increases while the Arduino operates and is only set back to zero, when the Arduino is reset.

The variable `timeDuration`, which we are interested in, is therefore measured as a difference between the current absolute time and the time at which the last change occurred. Looking at the Fig. 3.9, the red dot represents the first voltage measurement made after the signal has changed from High to Low. The time of the last High state has to be measured and it is equal to `millis() - timePrev`, where the `timePrev` in this case is the time at which the High state has begun.

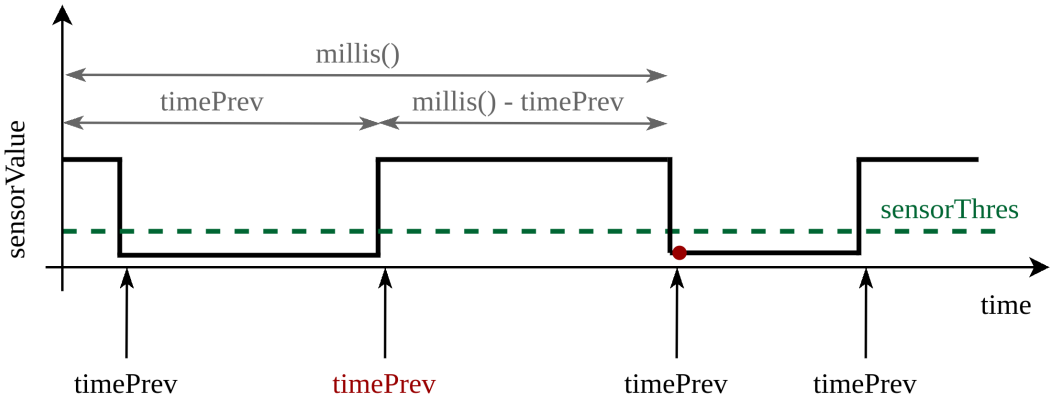


Figure 3.6: Measuring the time durations.

The identification of the last finished state as High or Low is made by entering the correct `if()` condition, and then accordingly we print on the Serial Monitor either `"H"` or `"L"`, followed by the current value of the variable `timeDuration`.

The corresponding flag needs to be raised. If we entered the High state, the `prevSignal` has to be set to true and if we entered the Low state, the `prevSignal` has to be set to false.

Finally, it should be noted that entering the `if()` conditions happen only at changes between Low and High states. Outside of this changes, the `loop()` function simply keeps measuring the `sensorValue`.

Blackout zone explanation

Not everything can be perfect. Imagine a situation when the currently measured signal was High and while it was supposed to turn back to Low some additional

light appeared in the surroundings that wasn't coming from the LED. If the boundary between High and Low states were touching, the graph of the measured `sensorValue` might for a short period of time become this:

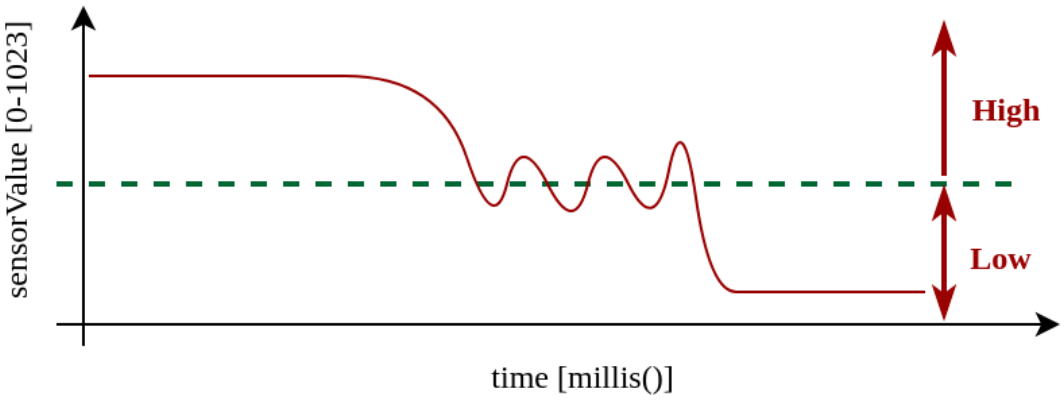


Figure 3.7: Signal corrupted due to light coming from different sources.

and give a nonsense measurements of short oscillations between High and Low state.

In order to protect the **receiving** phase we decided that we won't make the boundary between High and Low state touch, therefore making a relatively big separation between what is interpreted as High or Low state. In that case even if the oscillation happens, the measurements inside the blackout zone will be ignored by Arduino.

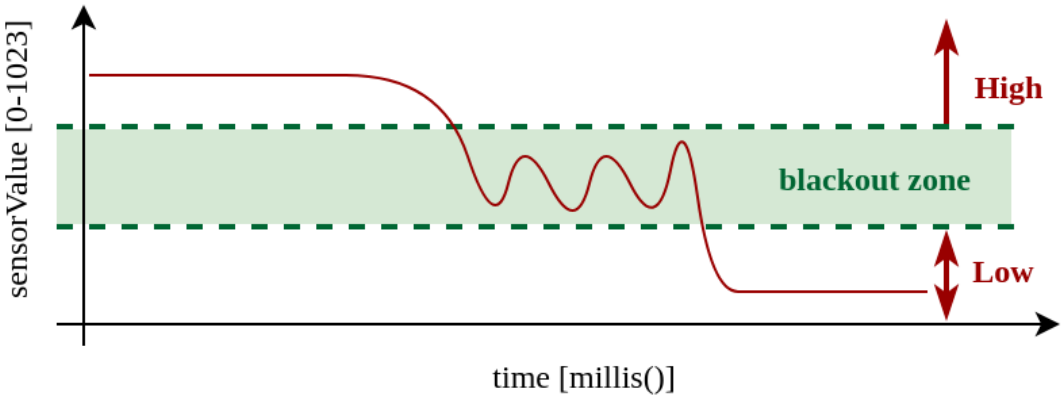


Figure 3.8: The signal inside the blackout zone is ignored.

3.2.3 Interpreting the Arduino output

The output from the Arduino device is being stored in a special file under Linux directory:

```
/dev/ttyACM0
```

As long as Arduino is measuring, we are receiving the outputs one by one in the form like this:

```
L 200
```

```
H 200
```

```
L 600
```

While this outputs are coming we're going to access the file `/dev/ttyACM0` to capture each one of them and our code is going to store and make sense out of that data.

The translation of the Arduino output to the actual stream of Morse signals ready to be interpreted by the `morse.cpp` code is handled by the file `arduinoReceive.cpp` (that's a lot of translations, n'est pas?).

Vector `durations`

The goal of the `arduino::read()` function is to build a vector called `durations` in which at even indices are the durations of High signals (in *ms*) and at odd indices are the durations of Low signals (in *ms* as well).

For instance, the Arduino output like this one:

```
L 200
H 200
L 600
.
.
.
```

will become a stream of consecutive durations like this:

```
{0, 200, 200, 600, ...}
```

Since this example output started with the Low signal we cannot place in at the first available index (index 0), because the Low signals are placed at odd indices. In that case a duration of 0ms is placed first (at index 0) and then the proper Low signal duration at the next index 1. These "zero" durations are simply placeholders and they do not spoil the received message.

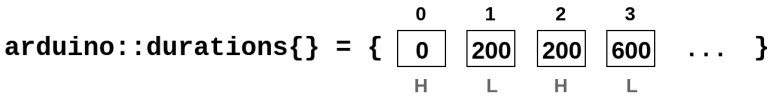


Figure 3.9: Constructing the vector `durations`.

Correct placement inside the vector `durations` is taken care by the following two conditions:

```
1  if (durations.size()%2 == 0)
2  {
3     switch(tempChar)
4     {
5        case "H":
6        {
7           durations.push_back(tempInt);
8           break;
9        }
10       case "L":
11       {
12          durations.push_back(0);
13          durations.push_back(tempInt);
14          break;
15       }
16       default:
17       {
18          std::cout << "Transmission error.";
19          break;
20       }
21    }
22 }
```

The first one is entered when the size of the vector is divisible by 2, meaning the index at which we'll be appending the next duration is also divisible by 2. When the current output from Arduino is a High signal ("H") we simply append to the vector because high signals can be placed at even indices. If it happens that the current output from Arduino (for whatever reason) is Low ("L"), we're going to first append the "zero" placeholder and then the duration of the incoming Low signal.

```
else
{
   switch(tempChar)
   {
      case "L":
      {
         durations.push_back(tempInt);
         break;
      }
      case "H":
      {
         durations.push_back(0);
         durations.push_back(tempInt);
         break;
      }
      default:
      {
         std::cout << "Transmission error.";
         break;
      }
   }
}
```

The second condition is handling everything else, so the cases when the current appending index is odd. The situation is now reversed - if the incoming Arduino signal is Low ("L"), we simply append it to the vector and if the incoming signal is High ("H") we first place the "zero" placeholder and then the proper signal duration.

The matter of trust

The final function in the `arduinoReceive.cpp` is `arduino::output()`. It's main role is to turn the Arduino output into a string of Morse alphabet characters, namely turn something like this:

```
H 600
L 200
H 200
L 200
H 600
L 200
H 600
L 600
H 600
L 200
H 600
```

```
L 200
H 600
L 600
```

into a string like this:

```
"-.-- ---"
```

Once we have the string written in Morse we can pass it on to the function `morse::outputText()` which will take care of the translation to text!

First of all, we have to know what does a duration of - say `H 200` mean - is 200ms a dot or a dash? Being a human, you probably didn't have problems seeing which High signals were dots and which were dashes in the message above. To find out you looked at the whole context to see what is the lowest and what is the highest duration of the High signal.

Our program will have to do something similar - it will look at the whole message first to figure out which durations correspond to the dot and dash.[1]

Now that the Arduino output is nicely sorted inside the vector `durations`, we're going to make use of knowing where the High and Low signals sit in that vector.

Because the message might, in a particular case, contain much more dots than dashes (or in reverse), we've decided that we will base our calculation of durations on the most frequently occurring one of the two.[2] It's simply the matter of trust, since the Arduino might measure them with a certain error.

First we calculate what is the average duration of the High signal `averageHigh`:

```
1  for (i = 0 ; i < durSize ; i += 2)
2  {
3     if (durations[i] != 0)
4     {
5        ++numNonZero;
6        averageHigh += durations[i];
7     }
8  }
9
10 averageHigh /= numNonZero;
```

[1] Notice that since we allow the message to be transmitted at different speeds (chosen by the user), we cannot make for example the `dotTime` duration fixed in the code.

[2] There are of course some extreme cases where the whole message broadcasted is the letter "e" or "t". We have no way of knowing what was the transmitted character. If this is the message that you would like to transmit, we encourage you to additionally write something like "Kisses and hugs.".

This for loop goes over every second element of the vector `durations` starting at the zeroth index, knowing that the High states are placed at even indices. We only count the non-zero durations, so if any "zero" placeholders were included in the vector they will be ignored.

Based on that we're measuring how many dots and how many dashes there are in the message and compute the average duration of a dot and a dash. We assume that any High state that lasted less than the `averageHigh` is a dot and any that lasted longer is a dash.

```
for (i = 0 ; i < durSize ; i += 2)
{
      if(durations[i] != 0)
   {
      if (durations[i] < averageHigh)
      {
         ++ numDot;
         averageDot += durations[i];
      }
      else
      {
         ++ numDash;
         averageDash += durations[i];
      }
   }
}
```

Additionally, this for loop calculates the total number of dots (`numDot`) and dashes (`numDash`) in the message.

If there is more dots than dashes we build the durations based on the duration of a dot:

```
if (numDot > numDash)
{
   averageDot /= numDot;
   bound13 = 2*averageDot;
   bound37 = 6*averageDot;
}
```

In the reverse case, we trust the duration of the dash signal more:

```
else
{
    if (numDash == 0 ) {return "";}
    averageDash /= numDash;
    bound13 = 2*averageDash/3;
    bound37 = 2*averageDash;
}
```

Perhaps a bit bizzare invention are the variables `bound13` and `bound37`. They are the auxiliary boundary times for dealing with Low states. They separate what we interpret as a space between Morse characters and space between words.

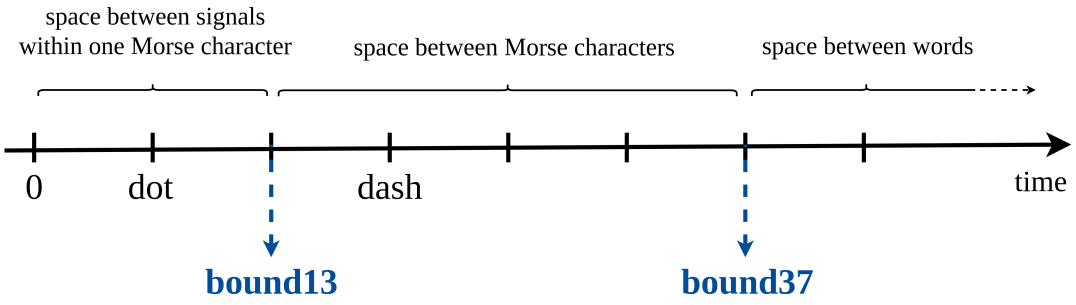


Figure 3.10: The explanation of the `bound13` and `bound37` variables.

They are always calculated in a way that `bound13` sit half-way between the dot and the dash duration and that `bound37` is six times longer than the dot duration.

The last for loop is assembling the whole message string by appending each signal into the variable `messageReceived`. Inside there is an if statement which checks if we're reading the element from the vector `durations` at the even or odd index.

For even indices we have therefore these two possibilities:

```
if (element < averageHigh) messageReceived += ".";
else messageReceived += "-";
```

This element can either represent a dot or a dash. Any High signal duration that is less than `averageHigh` will be interpreted as a dot. Anything longer will be interpreted as a dash.

For odd indices we have:

```
if (element > bound13)
{
   messageReceived += " ";
   if (element > bound37) messageReceived += "   ";
}
```

If the low signal was anything longer than `bound13` we're going to have to append at least a space between Morse characters but if additionally the low signal was longer than `bound37` it's a space between words.

Notice that the case when `element < bound13` represents the space between Morse signals within one Morse character, therefore we don't have to append anything specific to the `messageReceived` in this case.

At the very end, the function `arduino::output()` returns the string `messageReceived` written in the Morse alphabet which corresponds to the message that was captured by Arduino.

Chapter 4

Putting it all together

Now comes the moment we've all been waiting for. It's time to put the two phases: **broadcasting** and **receiving** together and see how the whole project is working in all of its beauty!

The picture below shows our complete setup of the *Objectif Morse*.

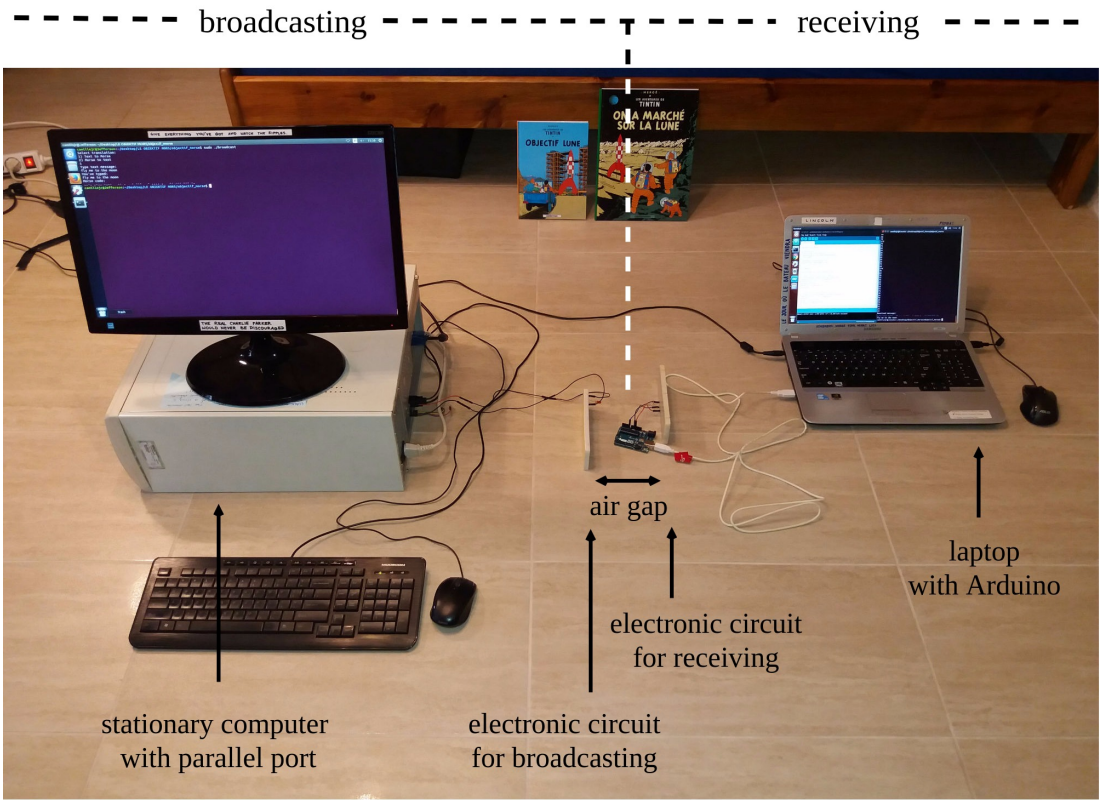


Figure 4.1: *Objectif Morse* complete setup.

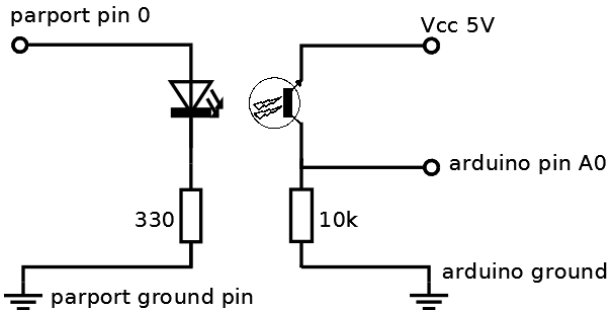


Figure 4.2: Scheme of the electronic circuits put together.

We have used two breadboards to completely separate the two electronic circuits. This allows for a little bit more debugging of the project because you can easily change the distance between the LED diode and the phototransistor. But of course you can build the two circuits on one breadboard too. You can also create a cardboard box to hide the LED-phototransistor couple - this will make the secret messages more secret as well as make the project work better in the brighter surroundings.

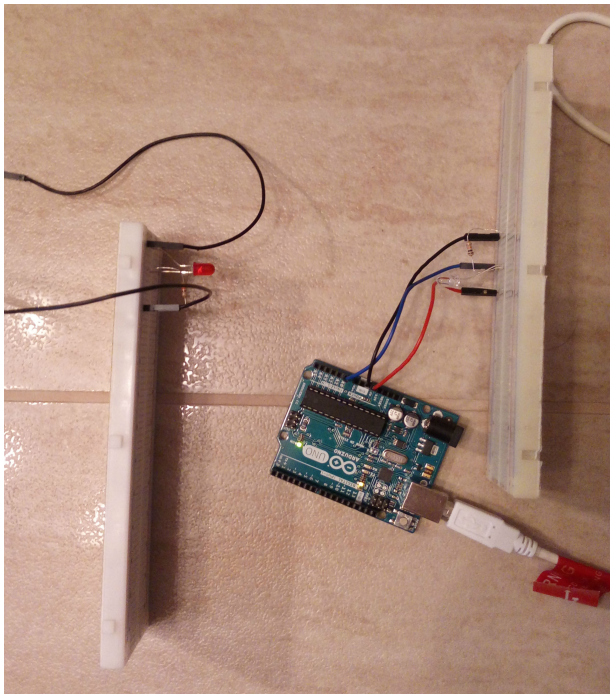


Figure 4.3: Electronic circuits put together.

Chapter 5

Post-credit scenes

This last chapter is a collection of post-credit scenes where we present a few of our ideas of exploiting the machinery that we have created.

5.1 It doesn't end here

Reading secret alien messages

Now it's your time to have some more fun! Forget about the **broadcasting** part, take the flashlight and try to create the message yourself by shining on the phototransistor. Hopefully you'll do better than us!

How far can it go?

Approximately 22cm separation was the maximum distance without transmission errors that we've observed during a daylight of a gloomy day, with closed window shades and with no artificial light in the room.

How fast can it go?

The parameter `dotTime` alows you to control the speed of the transmission. Feel free to play around with it! We've observed that `dotTime < 2000` microseconds is when the transmission errors begin to occur. It is also dependent on the distance between LED diode and the phototransistor and on the general luminosity of the surroundings.

Sending a picture

Can you imagine how ineffective it would be to transfer images between computers using Morse?

Light to sound

At some point we realized that it won't be a lot of effort to replace light Morse transmission with sound Morse transmission! Using a microphone and a buzzer you can achieve pretty much the same thing.

5.2 Want to do more?

There was a lot more ideas and unfortunately not always enough time.

One day we would like to extend the project to transmit messages over larger distances. Perhaps use a laser beam and a solar panel? Could this be a new way of the internet-free communication between neighbours? Although quite susceptible to the man-in-the-middle attack... But what if neighbours used an optical fibre hidden in the ground?...

As Cliff Stoll once said [5]:

I'm interested in seeing where the curiosity will lead to, not: oh, where have we been.

We encourage you to explore, learn, make a lot of mistakes and the most important: have a lot of pleasure of finding things out. Perhaps we'll hear from you soon when you share your ideas with us!

Bibliography

[1] *Les Aventures De Tintin: Objectif Lune*, Hergé

[2] *Les Aventures De Tintin: Le Lotus Bleu*, Hergé

[3] C++ reference

[4] `ioperm`, Linux Programmer's Manual

[5] Cliff Stoll: Good Science