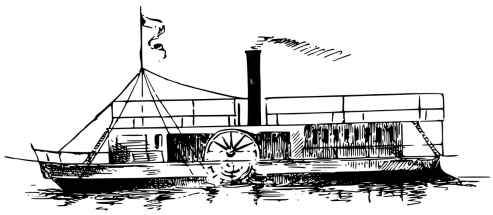
Objectif Morse

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

J. Aleksanderek
K. Zdybal

October, 2016 - September, 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:
jakubaleksanderek@gmail.com
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 work that we did, the ideas and the solutions that we came up with and of the things that we have learned.

The second purpose is to serve as a tutorial for you, if you ever feel like embarking on the same adventure as we did. And it is not very difficult! Following the document, you will be able to (hopefully) accomplish the same mission or 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 requried as a prerequisite, or a strong will to invest time and effort to learn many new things as you go.

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

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 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 after reading the Arduino output and printed again 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* (eng. *Destination Moon*), and is hence a tribute to our initial inspiration.

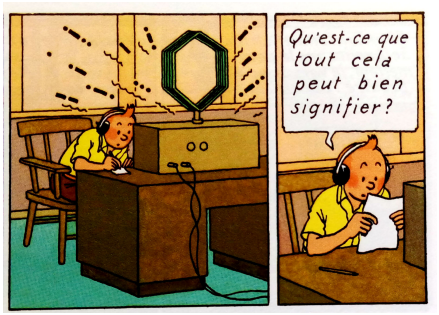


Figure 1.1: Excerpt from *Les Aventures de Tintin: Le Lotus Bleu* by Hergé

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

Electronics:

1. LED blue diode
2. phototransistor
3. resistors: 330 Ohm, 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
4. of course you need some monitors and keyboards...

Optional:

1. flashlight - increases the interactivity of this project

1.4 Code used

The *Objectif Morse* project is coded in C++ language and we have utilised the most powerful part of C++: object-oriented programming.

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:

```
arduinoReceive.cpp
arduinoReceive.h
receiveMain.cpp

makeFileR
```

Arduino:

```
arduinoCode.ino
```

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 alphabet. This part introduces the message input using alphanumeric characters 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 on the parallel port (`sendToPort.cpp` and `sendToPort.h`).

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

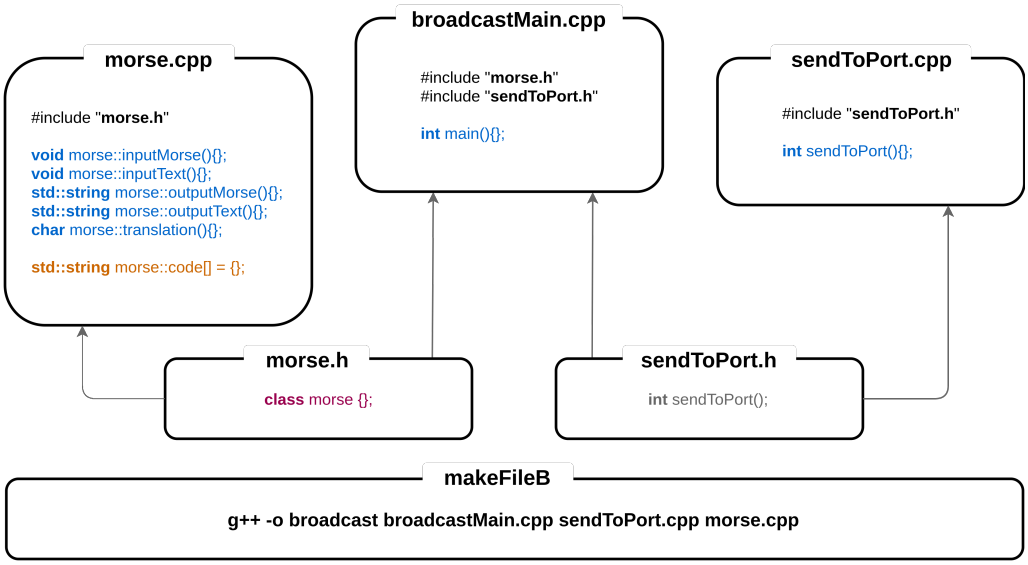


Figure 2.1: Code structure for **broadcasting** phase.

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 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);

};
```

We have the following 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 alphabet 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()` - a public function that simply returns the ready string consisting of message written in Morse alphabet.

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

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

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 that create the Morse alphabet table. 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 Fire it up using 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 in the command line:

```
make -f makeFileB
```

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

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

```
./broadcast
```

2.1.6 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 it!

As long as the electronics isn't connected to the parallel port, when you select 1, there will be 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.
```

2.2 How does it work

This section is diving a little deeper into explanation of ideas behind the **broadcasting** phase and it also tells you how to set up the electronics part.

2.2.1 Parallel port connection

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:

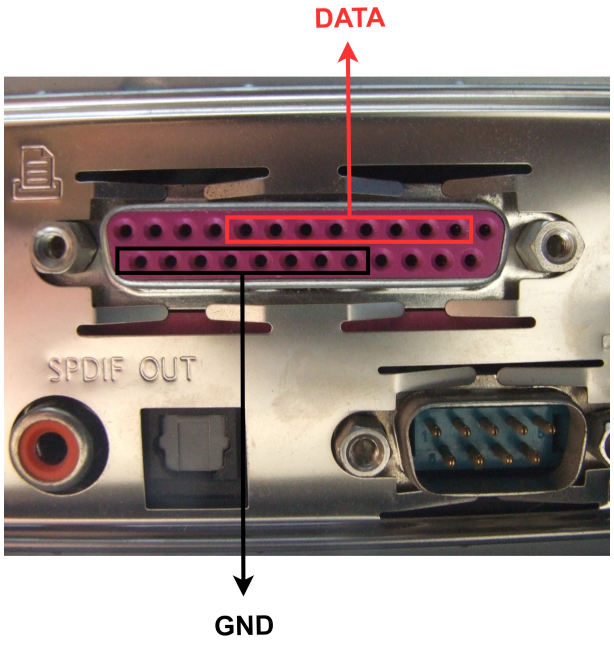


Figure 2.2: Parallel port pins: DATA and GND.

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:

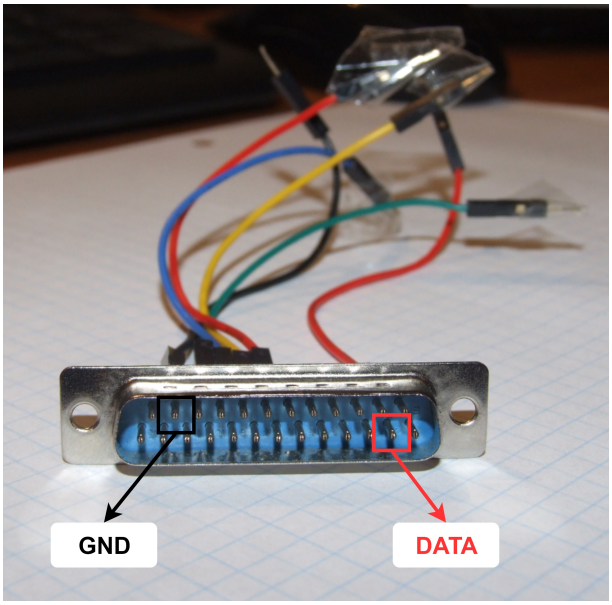


Figure 2.3: Parallel port plug.

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.

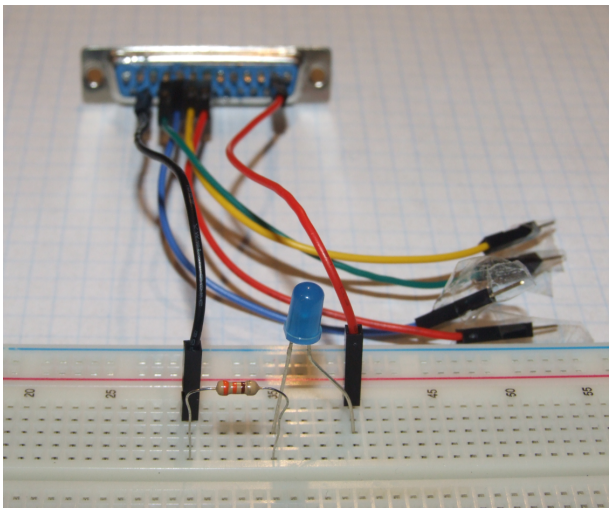


Figure 2.4: Electronic circuit for broadcasting.

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:

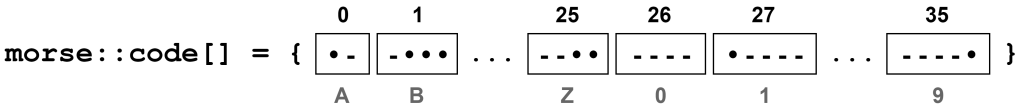


Figure 2.5: Morse code array.

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:

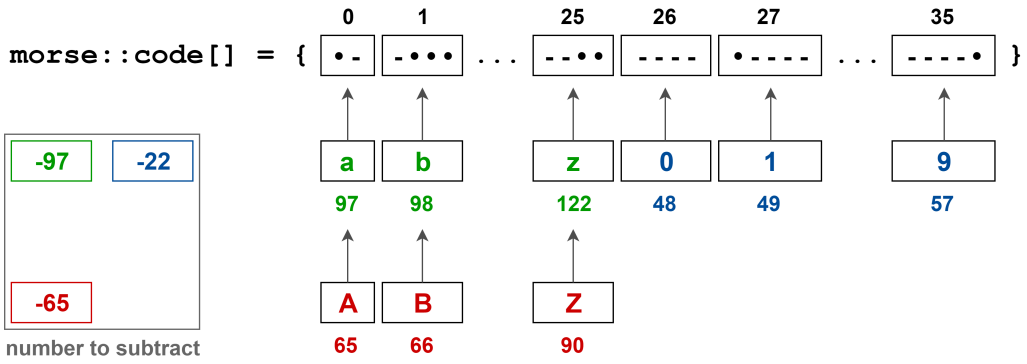


Figure 2.6: Mapping alphanumeric characters to the morse code array.

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

In this part we describe the contents of the Arduino code.

The aim of the Arduino code

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 400
```

Which has the interpretation as follows: Low state lasted 200 *ms*, High state lasted 200 *ms*, Low state lasted 400 *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, which probably should be thought over...

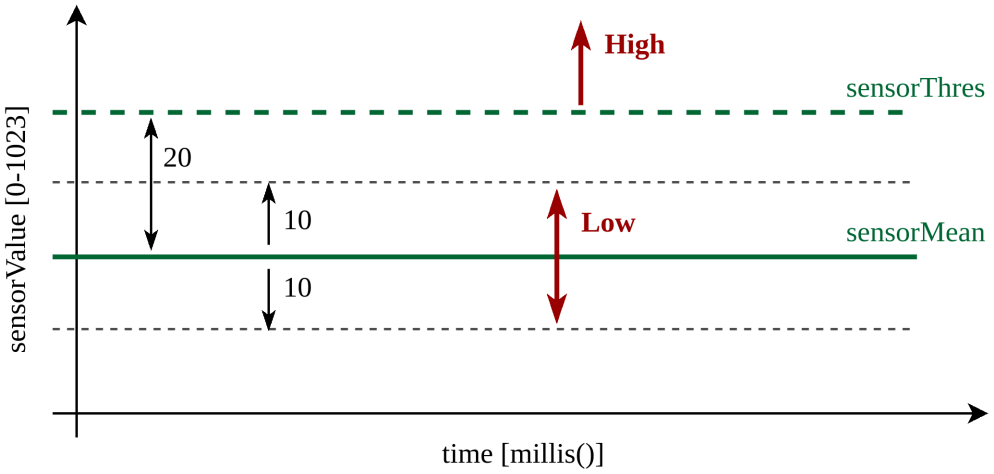


Figure 3.1: Interpretation of Low and High states using defined variables.

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 day (or night, or room) and what part will be changes due to the LED diode shining.

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).

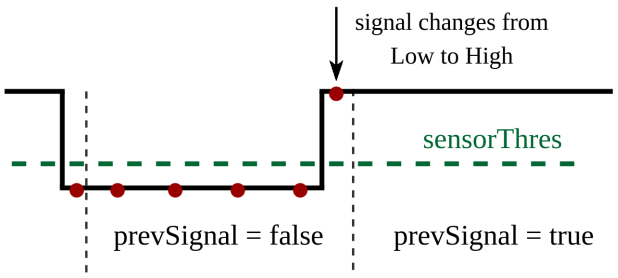


Figure 3.2: Signal changing from Low to High.

The signal is changing from High to Low when the voltage read is in-between 10 from the `sensorMean` value and when the previous signal was High (when the `prevSignal` is set to true).

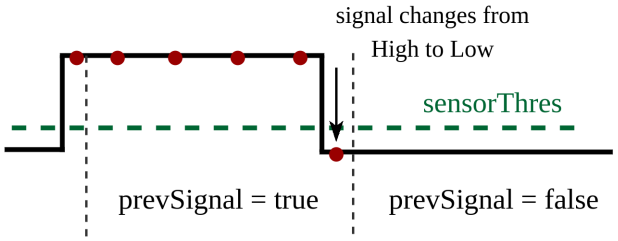


Figure 3.3: Signal changing from High to Low.

Measuring the time durations

Each time we enter one of the two `if()` conditions, the time of the last finished state (High or Low) must be measured, identified 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 occured. Looking at the Fig. [3.4], 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 (marked red).

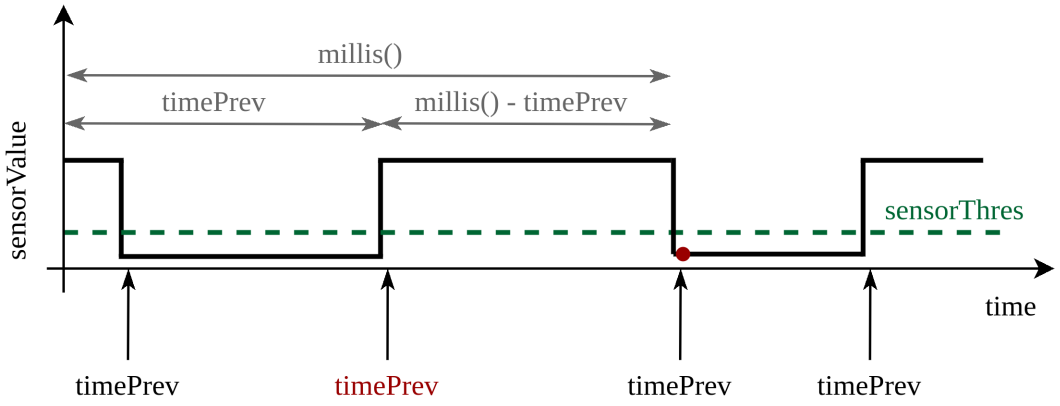


Figure 3.4: Measuring the time durations.

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

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

Chapter 4

Putting it all together

Include a picture with a full setup!

Chapter 5

Remarks

- reading secret alien messages

- laser beam

Bibliography

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

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

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