Chapter 2

Proposed Solution

In Chapter 1 the problem and the goals for this project were described. The idea around the solution is to build a motor network that can be placed in a prototype and be tested. To do so it is needed at least a micro-controller, vibration motors and a graphical interface. In the following sections is an explanation of what is used to fulfill the goals set in the beginning.

2.1 Hardware

2.1.1 Arduino MEGA

To control the motors a micro-controller is needed. The choice made was to use an Arduino.

Arduino is an Italian open source computer hardware and software company that produces prototyping and testing boards. Arduino board designs use a variety of microprocessors and controllers, the most usual being ATMEL micro-controller. The boards have both digital and analog input/output (I/O) pins that may be expandable by stacking shields and other circuits to them. These boards feature serial communications interfaces, including Universal Serial Bus (USB) which are also used for loading programs from personal computers. It is a very affordable solution and also easily expandable[38].

There are several Arduino available, such as Arduino UNO, Arduino Micro, Arduino Nano, Arduino MEGA Arduino DUE, Arduino Leonardo and more.

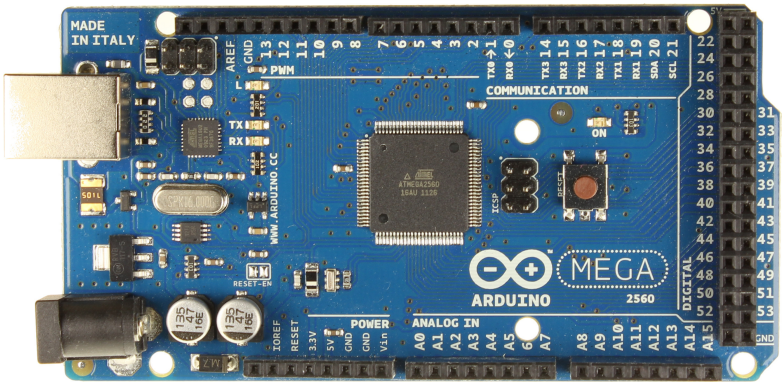

Figure 2.1: Arduino MEGA 2560 [39]

On Table 2.1 there are some important characteristics of Arduino MEGA 2560.

The biggest reason to chose MEGA over other Arduino is the fact that, besides Arduino DUE, MEGA is the μC with the most digital IO pins (both with 54). However, since the MEGA operating voltage is 5V and the DUE is 3.3V [40], Arduino MEGA was selected.

Table 2.1: Arduino MEGA 2560 characteristics [39]

Micro-controller	ATMega2560
Operation Voltage	5V
Input Voltage	7-12V
Digital I/O Pins	54 (15 PWM output)
Analog Input Pins	16
Flash Memory	256KB (8KB used by bootloader)
EEPROM	4KB
SRAM	8KB
Clock Speed	16 MHz
Dimensions	101.52x53.3mm
Weight	37g

2.1.2 Vibration Motors

As it was said in Chapter 1 one of the main goals of this project is to make a control system of a network of vibration motors.

There are several vibration motors in the market, such as Eccentric Rotating Mass (ERM) motors (in which the weight found on the rotating vibration motors are what produces the vibration), Brush-less Direct Current (BLDC) motors (this motor does not have brushes) and Linear Resonant Actuator (LRA) motor (a unique type of vibration motor with its performance similar to a speaker) [41].

The motors used in this project are ERM motors, more specifically DC brush motors (permanent magnet and brushes).

Brushed DC motors are widely used in applications ranging from toys to push-button adjustable seats. A standard DC brush motors consists on: [42]

- Stator
- Rotor
- Brushes
- Commutator

The construction of a standard brushed is shown in Figure 2.2.

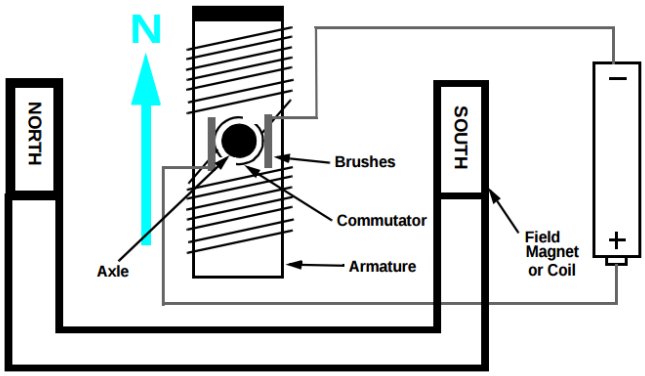

Figure 2.2: Simple Two-Pole Brushed DC Motor [42]

Inside the ERM motors, there are also various vibration motor forms, as shown in Figure 2.3.

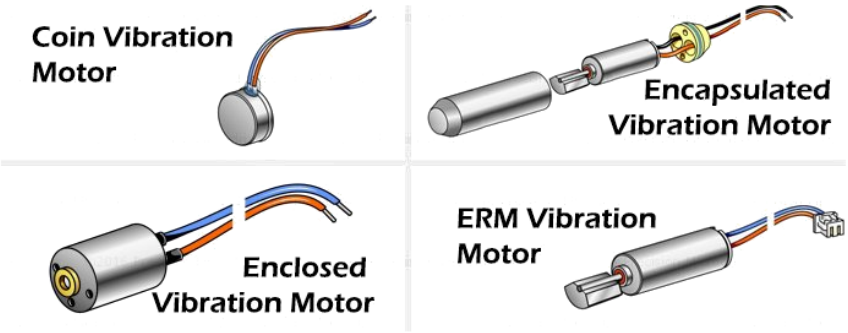

Figure 2.3: Vibration Motor Form Factors [41]

Since the coin vibration motor (Figure 2.4) is compact and easy to use it was the chosen for this project.

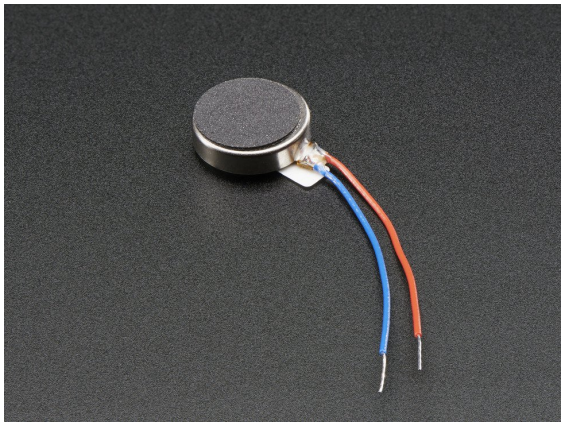

Figure 2.4: Mini DC Motor [43]

On Table 2.2 some important characteristics of this motors can be seen.

Table 2.2: Mini DC Motor Characteristics [43]

Voltage	2V-5V
Diameter	10mm
Thickness	2.7mm
5V current draw	100mA
4V current draw	80mA
3V current draw	60mA
2V current draw	40mA
Weight	0.9g

Motor Functioning

On Figure 2.5 there is one disassembled motor.

Figure 2.5: Motor Disassembled

Typically, a ERM motor is a DC motor with an offset (non-symmetric) mass attached to the shaft. In the case of the coin vibration motor, the non-symmetric mass is inside the motor capsule.

The stator inside the motor generates a stationary magnetic field that surrounds the rotor. The rotor (or armature) is made up of the windings. When these windings are energized they produce a magnetic field, and when the magnetic poles of the rotor field start to be attracted to the opposite poles generated by the stator, the rotor starts to rotate [42].

As the motor rotates, due to the fact that the mass center of the non-symmetric mass is not in the physical center, there is a net centrifugal force, which causes a displacement of the motor. As the motor is constantly being displaced, that is perceived as vibration.

On Figure 2.6 there is a scheme by Precision Microdrives that explains how the motor is built.

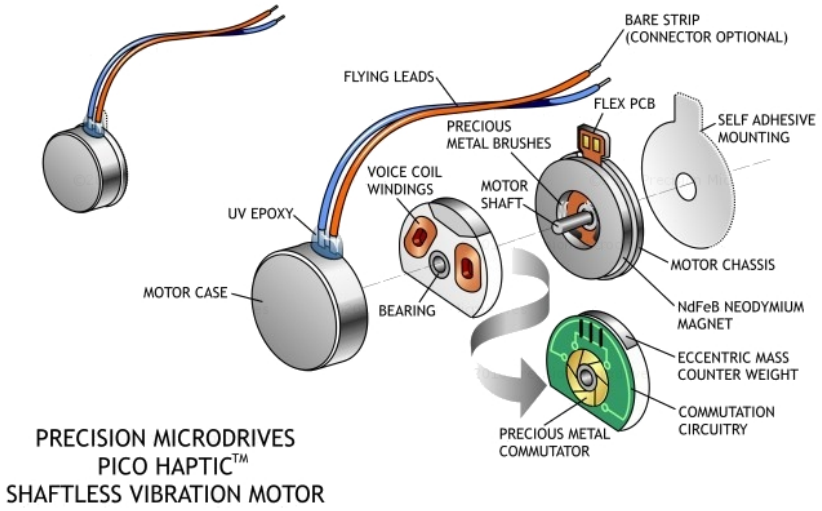

Figure 2.6: Scheme of the Motor [44]

Voltage Control

A Pulse Width Modulated signal is a type of digital waveform. It alternates between bursts of high (on/1) and low (off/0) at a fixed frequency. The difference between PWM signal and other digital signals is the fact that the time that the signal is high (and therefore low) can be varied [45].

When we change the pulse width we are actually changing the average voltage allowing any value between zero and the maximum voltage to be represented by increasing or decreasing the ON pulse width.

The PWM signal has three separate components:

- A voltage, V_{PWM} - the value of the ON or high voltage level (typically between 2 5V).
- A Frequency - the period of one clock cycle (one high pulse and one low pulse).
- A duty cycle - the ratio of ON/OFF time.

The average voltage of a PWM wave can be calculated using Equation 2.1.

$$V_{avg} = V_{PWM} * DutyCycle \tag{2.1}$$

For instance, if we have a 5V PWM signal (Arduino MEGA has a 5V output signal) with a duty cycle of 50%, then the average output voltage is calculated according to Equation 2.2.

$$\begin{aligned} V_{out} = V_{avg} = V_{PWM} * DutyCycle \\ = 5V * 50\% \\ = 2.5V \end{aligned} \tag{2.2}$$

This is the voltage that the motors will receive.

2.2 Software

2.2.1 ROS

ROS (Robot Operating System) is a open source software used for the development of robots. Its philosophy is to make a piece of software that could work in other robots by making little changes in the code. ROS was originally developed in 2007 by the Stanford Artificial Intelligence Laboratory (SAIL) and in 2008 Willow Garage continued its development. Currently a lot of companies and research institutions use ROS for the development of their projects (such as the NAO robot) [46].

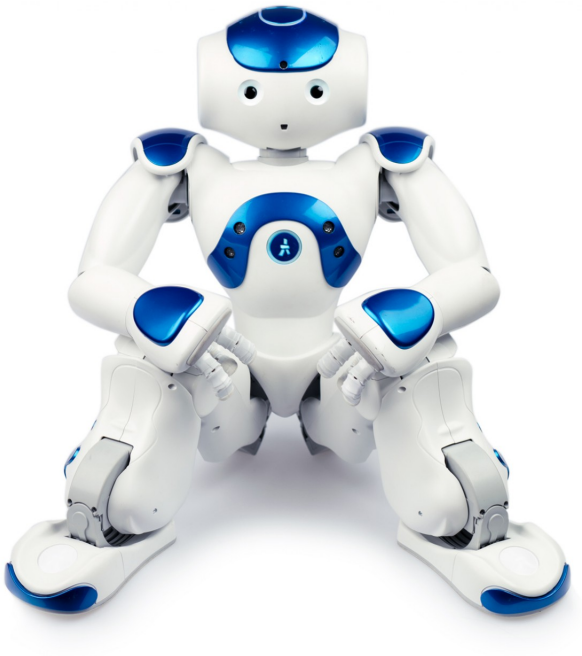

Figure 2.7: NAO Robot [47]

The sensors and actuators used in robotics have also been adapted to be used with ROS, and day by day, there is an increasing number of devices that are supported by this framework.

ROS provides standard operating system facilities such as hardware abstraction, low-level device control, implementation of commonly used functionalities, message passing between processes, and package management [46].

The main advantage of ROS is that nodes in ROS do not have to be on the same system (multiple computers) or written on the same language. This makes ROS very flexible and adaptable to the needs of the user. [48] That is the main reason why ROS was chosen over, for example, the creation of a shared memory. The communication between nodes can be done through messages or services. In this work, the method used is the message one. For this, a publisher and a subscriber to a topic must be created, in which the first publishes the message as the second receives it and acts according to it.

2.2.2 Rosserial

Rosserial is a protocol for wrapping standard ROS serialized messages and multiplexing multiple topics and services over a character device such as a serial port or network socket [49]. It allows electronic devices to talk directly to the rest of the ROS system using topics (sending messages or even services over a serial interface).

The Rosserial protocol is compiled just like any other ROS package and it contains several Client Libraries. The one used in this project is the *rosserial_Arduino*.

To be able to make the Arduino communicate with ROS, one must run the *rosserial_python* package and the *serial_node.py* node, mentioning the correct port where the Arduino is connected, just like the following, where *ACM0* is the port where the Arduino is connected [50].

rosrun rosserial_python serial_node.py /dev/ttyACM0

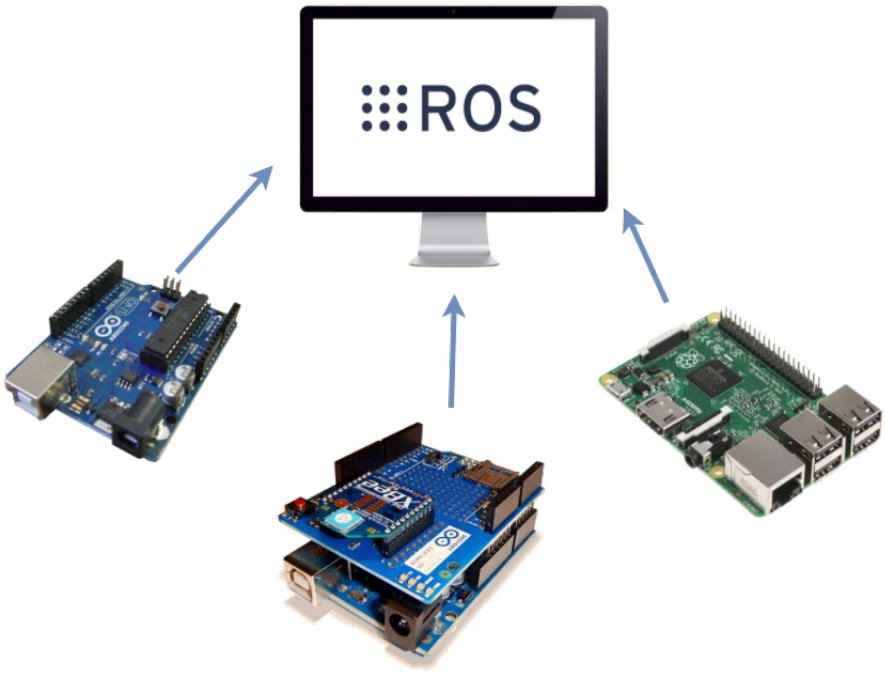

Figure 2.8: Rosserial diagram

2.2.3 Qt

Qt is a cross-platform application development framework for desktop, embedded and mobile[51] [52].

It is not a programming language but instead a framework written in C++. A preprocessor (MOC - Meta-Object Compiler) is used to extend the C++ language with features like signals and slots (mechanism used for communication between objects).

Before the compilation the MOC parses the source files written in Qt-extended C++ and generates standard compliant C++ sources from them.

Although any build system can be used with Qt, it brings its own qmake (that automates the generation of Makefiles). Even thought that qmake is the most used build system, CMake is also a popular alternative to build Qt projects.

Qt comes with its own Integrated Development Environment (IDE) named Qt Creator.

With Qt, GUIs can be written directly in C++ using its Widgets module. Qt also comes with an interactive graphical tool (Qt Designer) which works as a code generator for Widgets based GUIs. It can be used stand-alone but is also integrated into Qt Creator (what happened in this project).

2.2.4 Timers library in Arduino

A timer is a clock that controls the sequence of an event by counting in fixed intervals of time. It is used for producing precise time delay and measure time events [53]. It can be programmed by some special registers.

For the purpose that it is going to be explained in Section 3.2.2, for this project the use of timers and their features was necessary.

Arduino Mega 2560 controller is the Atmel AVR ATmega2560. As it can be seen in its datasheet [54] Arduino Mega has 6 timers: Timer 0, Timer 1, Timer 2, Timer 3, Timer 4 and Timer 5, with all of them being 16 bits except Timer 2 which works with 8 bits. The biggest difference between a 8bit and 16bit timer is the timer resolution, as 8bit means 256 values while 16bits means 65536 values for higher resolution or longer counter [55].

Since timer 0 is used for the software sketch functions like *delay()*, *milis()*, *micros()* and timer 2 only has 8 bits, those could not be used.

For this project two timers were needed. For reasons that will be explained in Section 3.2.2 timer 1 and timer 4 were selected and configured.

Obviously that configuration could be done by changing the timer registers by hand. However, there are already some libraries that do that for us. For this project, those libraries are *TimerOne* [56] and *Timer4* [57].

This libraries have the following functions [56]:

- $initialize(period)$: it needs to be called before any other. It initializes the timer with a desired period in μs (by default it is set at 1s).
- $setPeriod(period)$: sets the period in μs.
- $pwm(pin, duty, period)$: generates a pwm waveform on the specified pin.
- $setPwmDuty(pin, duty)$: sets the pwm duty cycle for a given pin if it was already set by calling $pwm()$ earlier.
- $attachInterrupt(function, period)$: calls a function at the specified interval in microseconds.
- $detachInterrupt()$: disables the attached interrupt.
- $disablePwm(pin)$: turns pwm off for the specified pin.
- $read()$: reads the time since last rollover in microseconds.

- *start*().
- *stop*().
- *restart*().

The minimum period or highest frequency this library supports is 1μs or 1MHz.

2.3 Components Integration

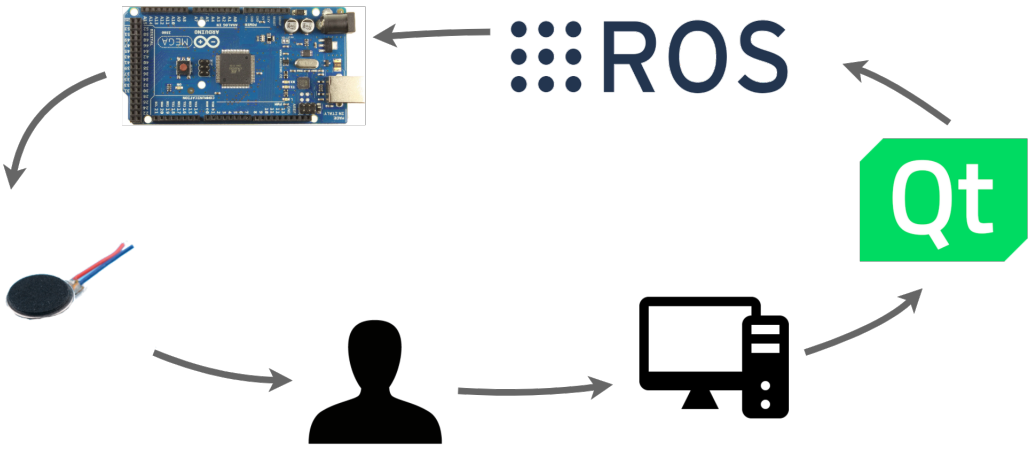

Figure 2.9: Diagram of all the components

Chapter 3

Development of an Application

This chapter displays all of the stages needed to develop the application. It explains the chosen location for the motors as well as the process of building a prototype while elucidating the organization of the program.

3.1 Peripheral nervous system and dermatomes

3.1.1 Placement of the Motors

The prototype built in this project was developed to be used in the arm/hand with vibration motors. As shown in Figure 3.1, the dermatomes that the motors need to focus on are C5, C6, C7, C8 and T1.

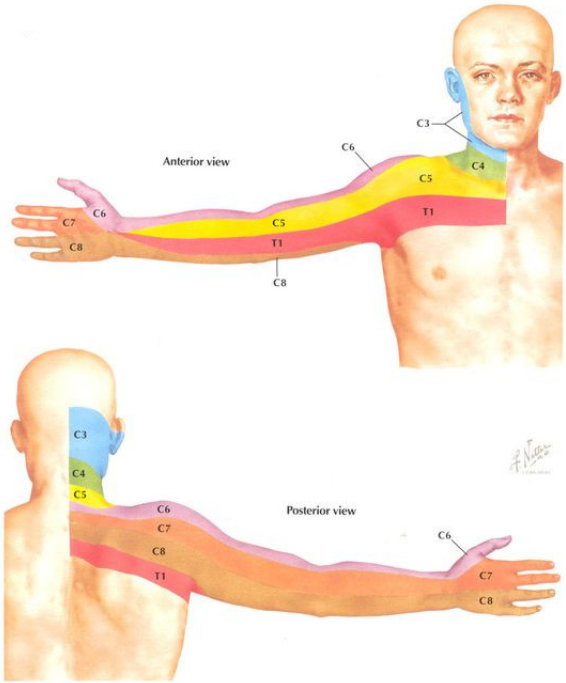

Figure 3.1: Schematic demarcation of dermatomes in upper body [34]

Considering this information, Figure 3.2 exhibits the proposed layout of the motors used in this project.

There are nine motors placed in strategical dermatomes places in the lower part of the arm/hand. On the upper part the other 7 motors were placed in the joints for diagnose purposes.

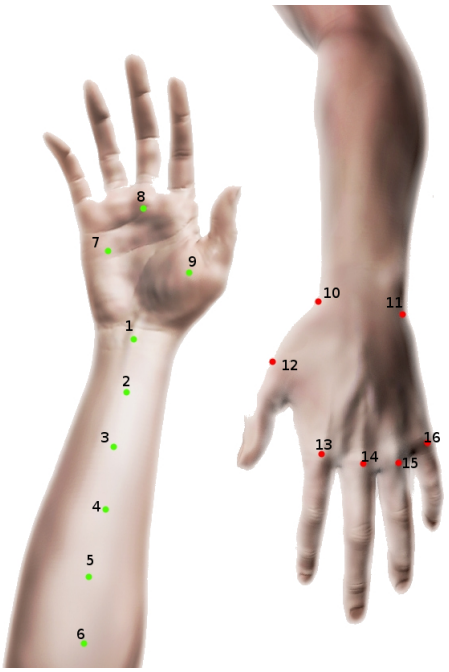

Figure 3.2: Placement of the motors

3.2 Motor Control

3.2.1 ROS nodes

One of the main goals of this dissertation work was to have a graphical interface so the user could control the motors.

Figure 3.3 shows a flowchart of the three nodes that where created to achieve that goal.

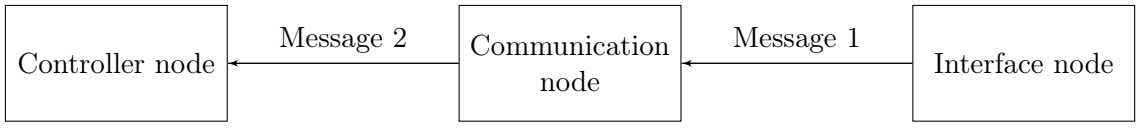

Figure 3.3: ROS nodes flowchart

The Interface node is a publisher node that sends information to the Communication node. This information is a message that contains the number of motors to turn on, their duty-cycle and their location (Message 1).

The Communication node connects the Interface and Controller node, being both a publisher and a subscriber. It receives the message from the Interface node, converts

the variables (more will be explained in detail in Section 3.4.1), thereafter sending the duty-cycle and the location of the motors to the Controller node (Message 2).

At last, the Controller node acts as a subscriber. It receives Message 2 from the Communication node and then proceeds to turn on the correct motors with the correct duty-cycle.

Figure 3.4: ROS nodes disposition made with *rqt_graph*

3.2.2 PWM manually generated

As it was stated in Section 2.1.1, Arduino MEGA has 15 PWM outputs. Yet, those are not enough to control all the motors, hence the PWM wave had to be manually generated using the available digital outputs from the Arduino. The solution found was to manually generate a PWM using timers and their interrupts.

An interrupt is an external event that interrupts the running program and runs a special interrupt service routine (ISR). After the ISR has been finished, the running program is continued [55]. According to the Arduino Mega data-sheet the priority of the different timers is different [54]. The priority order of the timers is expressed below.

1. Timer 2 (bigger priority).
2. Timer 1.
3. Timer 0.
4. Timer 3.
5. Timer 4.
6. Timer 5 (smaller priority).

Having in mind that two timers were needed, one with bigger priority than the other, and the fact that timer 2 is a 8-bit timer, timer 1 and timer 4 were selected.

- Timer 1: prime and slower timer with a frequency of 1kHz (period of 1μs)
- Timer 4: faster timer with a frequency of 10kHz (period of 0.1μs)

The idea to generate the PWM wave is to have the timer interrupt service routine generating the HIGH/LOW signal.

Figure 3.5 is a scheme for an easier comprehension. More about this PWM generation will be explained in the following Section 3.2.3.

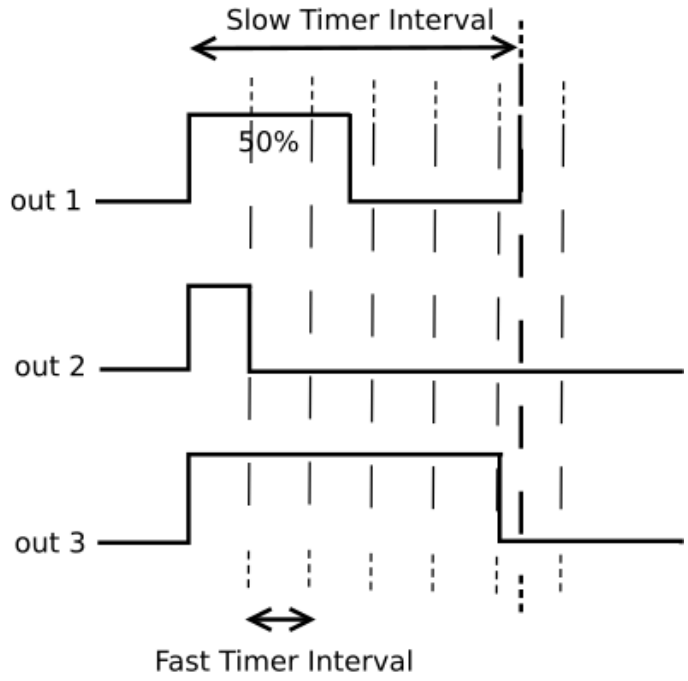

Figure 3.5: Timers behaviour

3.2.3 Controller

In Section 2.1.1 Arduino Mega was defined as the controller for this project and the controller node was identified as a communication node subscriber. The message received is the intensity and the created bitmask.

Figure 3.6 shows the main structure of the controller program.

Two ISR were created in the program setup to enable the creation of the PWM wave manually and turn the motors on and off with the correct intensity.

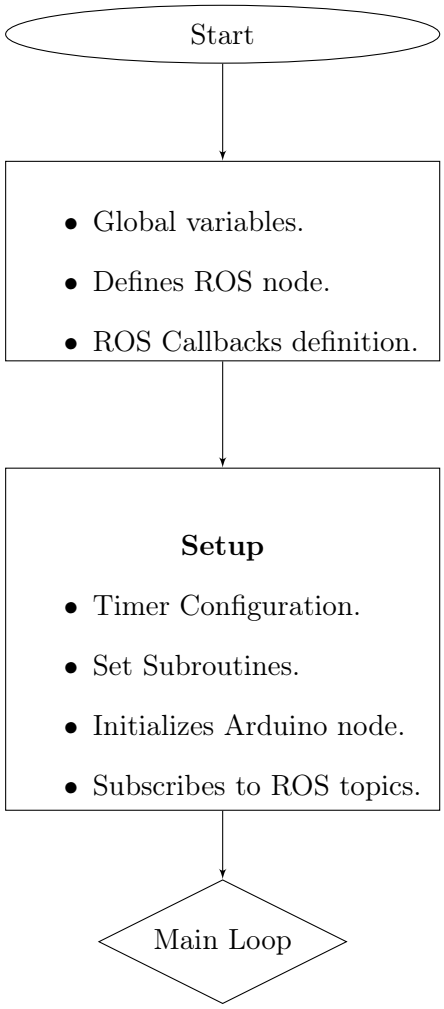

Figure 3.6: Controller flowchart

Every time timer 1 reaches the period it was configured with, the interrupt function checks using the motorMask variable whether or not the motor should be on. If the motor is supposed to be on the value HIGH is given to the correspondent motor pin. Figure 3.7 illustrates the Interrupt Subroutine related to timer 1.

When timer 4 reaches its period, it verifies if the desired duty cycle (intensity) has been reached or not, turning the motors off if it is supposed to (giving the motor pin the value LOW). Figure 3.8 shows the ISR related to timer 4.

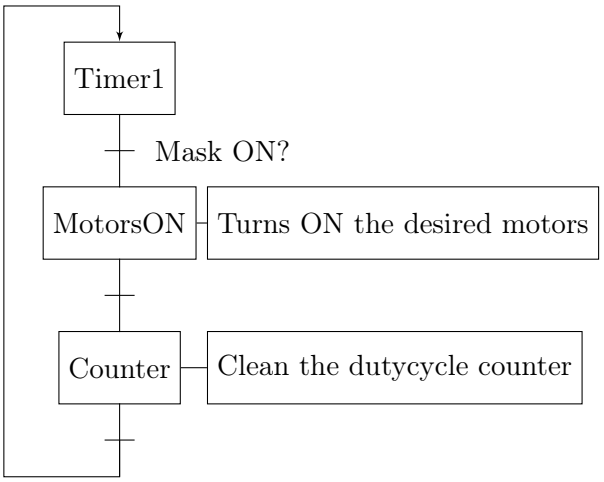

Figure 3.7: Interrupt Subroutine: Motors ON

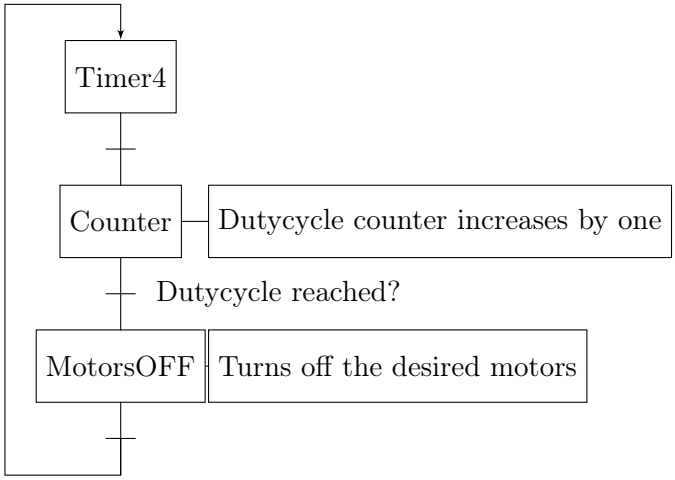

Figure 3.8: Interrupt Subroutine: Motors OFF

3.3 Graphical user interface

Previously the location of the motors and their intensity were identified as the variables that the user would be able to control. To allow an easier control of these variables a graphical interface was developed.

3.3.1 Packages and Classes

Due to the fact that the interface is a ROS node (*qtguinode* as seen in Section 3.2.1) a ROS package was created. This package is called *qtgui* and it is there that all the interface was built and the message was defined.

The design of the interface is made using a class that *QtDesigner* creates on its own: the *MainWindow* class. This is the class associated to the user interface.

ROS has a lot of information that it needs to work properly. It needs to be initialized, the ROS node needs to be named, a handler to that node needs to be created, among other things. To do so, the class *QtPublisher* was conceived inside the qtgui package. This class is responsible for the creation of the handler and to tell the master what type of message is going to be published and where. The topic *guichattergui* was defined as the topic where the message is published. Furthermore, the function *SendMessage* was also established. This function is responsible for broadcasting the message to anyone who is connected to the topic previously defined.

Besides this class, the *qt_ros_interface* package was also created. In here, the ROS initialization (*ros::init()*) is made and the node is named.

Figure 3.9 shows the organization of the interface node.

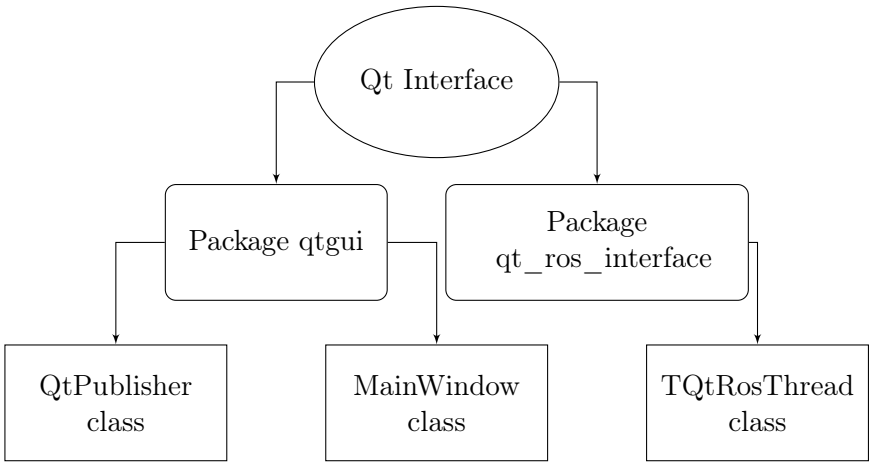

Figure 3.9: Interface organization

3.3.2 Message Created

The message created consists on the variables that the user can control (intensity and location) and the number of motors that are active. To this message name it was given the name *GUIDados.msg*.

```
uint8 numberOfMotors
uint8[ ] intensity
uint8[ ] location
```

Figure 3.10: Message GUIDados

Figure 3.10 illustrates the message that was created. The type of variable *uint8* (unsigned 8-bit int) was used due to the fact of being the smallest variable that ROS messages works with. It is the equivalent of the variable type *char* in ROS. While the number of motors is a simple number, both the intensity and location are an array of numbers. Since 16 motors are being used, the size of the array is 16.

3.3.3 Package Organization

Figure 3.11 shows how the package *qtgui* is organized.

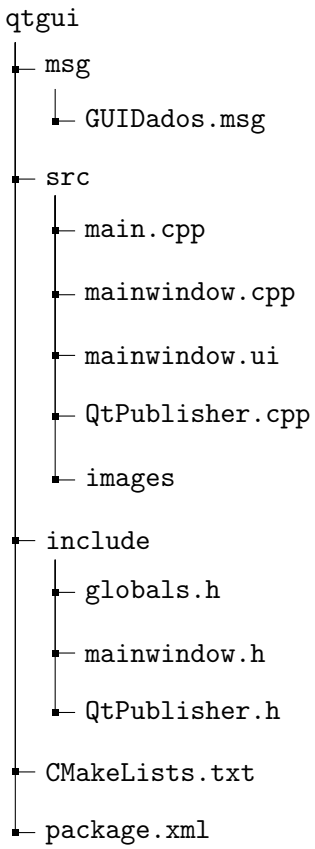

Figure 3.11: Organization of the qtgui package

3.3.4 CMakeLists

ROS is built using *catkin_make* and Qt with *qmake*. The first thing done was to change the built option from *qmake* to *cmake*, so it would be easier to mix the Qt CMakeLists and the ROS CMakeLists.

Since ROS is being used in the interface node, some important things such as the *catkin_libraries* and *QT_libraries* had to be added in the CMakeLists file. Important packages like *Qt4* and *qt_ros_interface* were also added.

3.3.5 User Interface

The interface offers the user two working modes:

- Motor control.
- Pattern.

When the interface is launched, the operating mode that appears on the screen is the Motor control (Figure 3.12). In this mode the user can select which motor to turn on and its intensity. It allows more than one motor functioning at the same time and it also has the option to activate and deactivate all the motors.

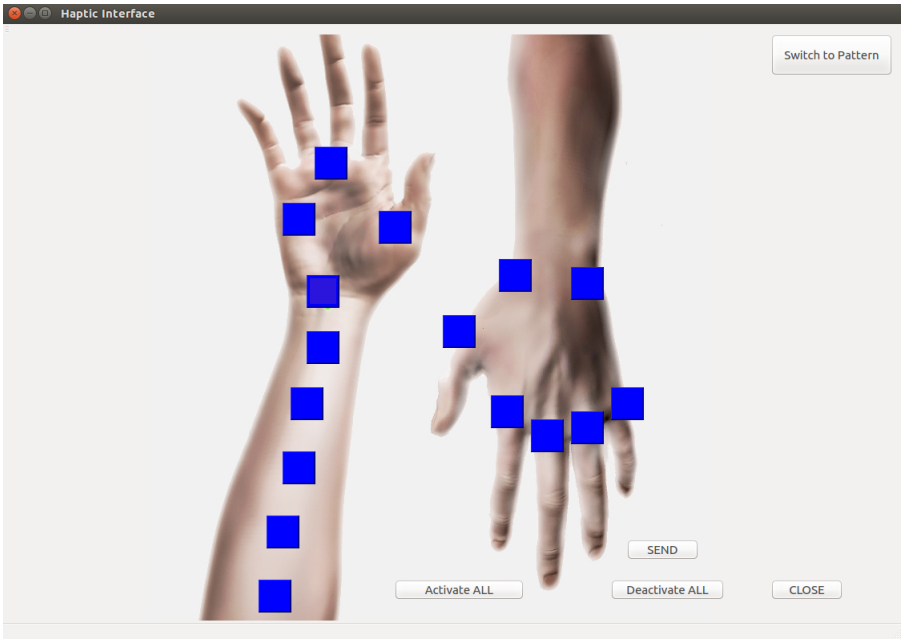

Figure 3.12: Interface: Motor Control

Initially the intensity selection box is not visible, however, every time a motor is selected (button is pressed) that option appears. The same happens with the activate all option. When the activate all button is clicked, all the motor's intensity selection box disappears and one general one emerges (therefore the chosen intensity is the same for every motor).

Another aspect in the motor control mode is the fact that, when the motors are OFF (buttons unselected) the buttons are blue and when the motors are ON the buttons turn red. This gives the user a visual help and was done to make the interface more user friendly.

To confirm which motors to turn on and their intensity the button Send should be pressed. What this will do is send the created message to the created topic as explained before.

One important point is the fact that, when the deactivate all button is clicked, the data is sent instantly. This happens, for instance, when the user selects motors to turn ON, besides having to give them a specific intensity, the operator usually does not want to turn ON only one motor but instead a group of them. Thus the information should only be sent when the user decides. Per contra, when the deactivate all button is pressed, the aim is to actually deactivate all the motors that are ON making it possible to send the message immediately.

When the button Switch to Pattern is pressed the window changes and so does the operation mode.

In the pattern option (Figure 3.13) the user can choose from previously made stimuli:

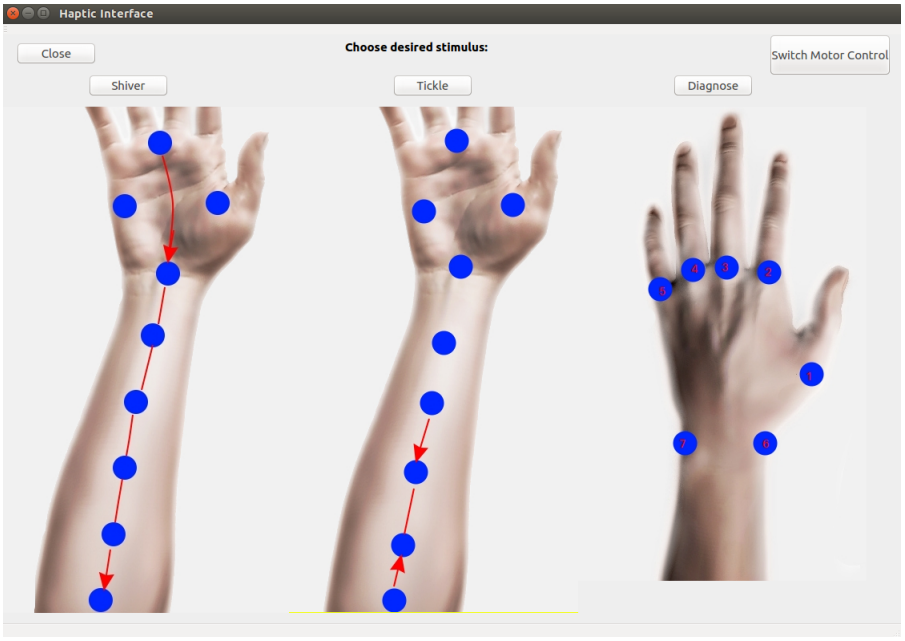

Figure 3.13: Interface: Pattern

- Shiver
- Tickle
- Diagnose

The shiver stimulus consists on turning the motors ON from the bottom to the top. That is done turning motor 8 on, followed by motor 1, 2, 3, 4, 5 and 6. The activation of the motors have an interval of 0.5 seconds. After turning one motor on, the previously activated motor is turned off so the shiver sensation is produced, like someone is moving their finger through the user arm.

The tickle stimulus dwells on the back and front movement. It starts on motor 6, goes to motor 5, 4 and 3 and then goes back to 4, 5 and 6 with a delay of 0.5s as well.

The last pattern stimulus created is the diagnose. For this stimulus, the motors used are the motors placed in the user joints. Those are motor 10, 11, 12, 13, 14, 15 and 16.

When the user selects the diagnose button an option for the delay between the motors and the intensity pops up. That delay can go from 1s up to 5s and the intensity can be either LOW, MEDIUM or HIGH. After pressing the OK button, motor 12 is activated, followed by motor 13, 14, 15, 16, 10 and 11. The delay and intensity button were added in the interface because, for medical diagnose, the time between the motors activation is important, and so is the different intensities. For instance, a higher intensity is easily felt by everyone yet a lower intensity is not.

To go back to the motor control option the user has to press the Switch to Motor Control button.

The process of associating the buttons and intensity boxes of the interface and the message to be sent is done using callbacks. Every time a button in the interface is clicked a callback is called giving values to the variables in the message.

If a given motor is selected, then in the location array its value is gonna be 1. If a motor is not selected then its value is 0. The same is done for the intensity array. The value of the intensity selection box is given to the intensity array in the motor position. For instance, if the motor 1 button is clicked and an intensity of 5 is selected (and assuming no other motor is active), then the variables are $numberOfMotors = 1$, $intensity[0] = 5$ and $location[0] = 1$.

When the Send button is pressed the *SendMessage()* explained previously is called and the message is then sent to the defined topic.

3.4 Communication

3.4.1 Communication node

Being both a subscriber and a publisher, the communication node receives the message from the interface node and processes that information to pass it over to the controller node.

The Communication node receives two *uint8* arrays with size 16 (intensity and location) and a *uint8* number (numberOfMotors).

The intensity array is composed by numbers from 0(none) to 10(higher). The location array is an array of 1s and 0s (1 if the motor is selected, 0 if the motor is not). To make the communication and the processing faster the size of the message is reduced. To achieve that reduction a bitmask that defines the motor behavior is generated.

```
for(int totalMotors=0;totalMotors<16;totalMotors++)
{
    uint8_t motorMask |= location[totalMotors] << totalMotors;
}
```

Figure 3.14: Creation of the bitmask that represents the motors to turn ON

For instance, if the motors to turn on are motor motor 1, motor 3 and motor 5, the bit mask has the value of 21.

While the message received from the interface node is an array with 16 *uint8* numbers (*uint8* is an unsigned 8-bit int therefore an array of size 16 has 128-bits), the message after the conversion is an *uint8* number with 16-bits. The message file created was named *Dados.msg*.

```
uint8[ ] intensity
uint8 location
```

Figure 3.15: Message Dados

After the conversion these two variables are sent to the controller node. The communication node package organization is represented in Figure 3.16.

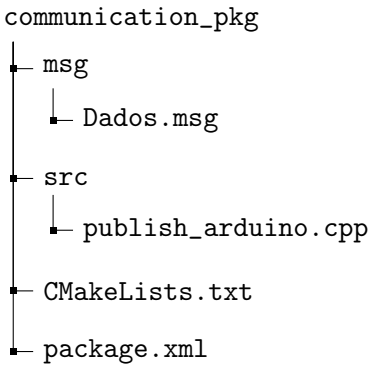

Figure 3.16: Organization of the communication package

3.5 Vibration Motors Sleeve

3.5.1 Creation of the Motors Sleeve

The created sleeve can be seen in Figure 3.17. The sleeve was made with Lycra and it has a fastener and holes for the user fingers to facilitate the wearing.

Figure 3.17: Prototype of the Sleeve

The motors were placed as it was stated in Section 3.1.1 and then sewn into the sleeve. To ensure a more robust prototype, nylon was used as lining (Figure 3.18).

Figure 3.19 represents the final prototype.

Figure 3.18: Inside of the Sleeve

Figure 3.19: Final Prototype

3.5.2 Connection of the hardware

In this project 16 motors are connected to the Arduino as the scheme in Figure 3.20 demonstrates.

To allow an easier connection of the motors to the controller a shield was developed using the Eagle CAD software. In this shield, 16 Arduino digital outputs are connected to one side of a 20 pins pinhead. To the other side of the pinhead there is a line of grounds. The Printed Circuit Board (PCB) was then printed at the Department of Electronics, Telecommunications and Informatics of the University of Aveiro.

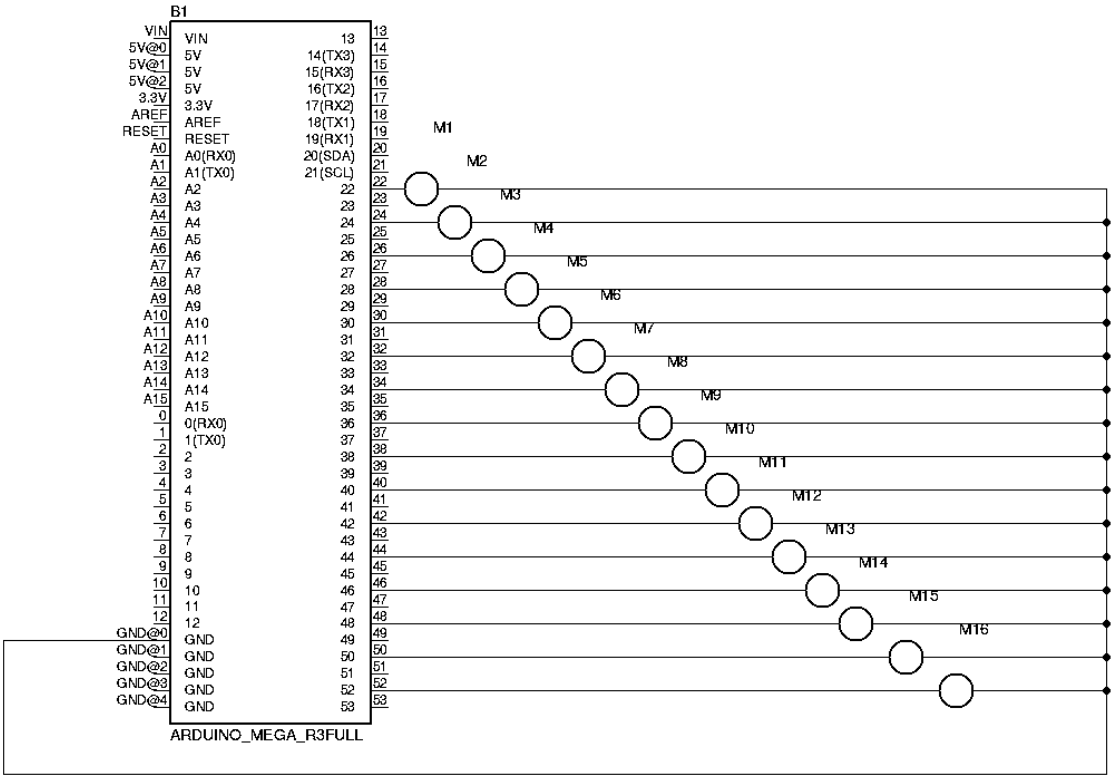

Figure 3.20: Electric Scheme

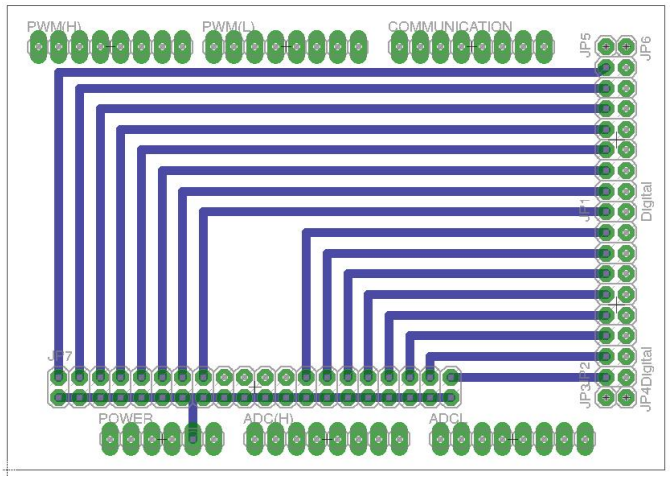

Figure 3.21: Eagle board design

Connected to the shield is a flat cable with a connector that fits into the pinhead. The motors are then connected to the other side of the flat cable.

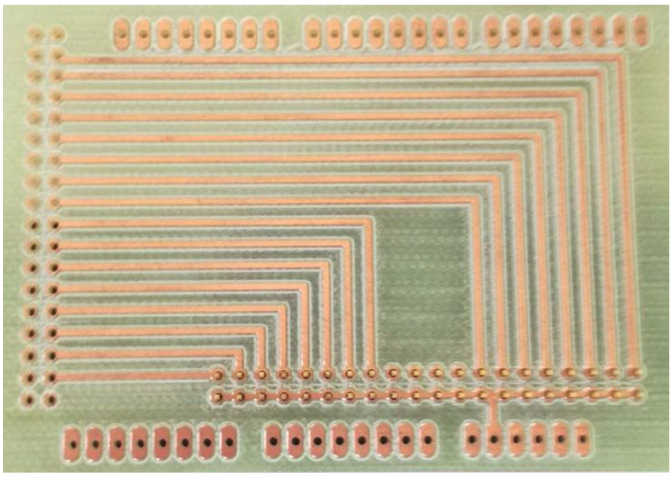

Figure 3.22: Printed shield

Figure 3.23: Shield connected to the Arduino

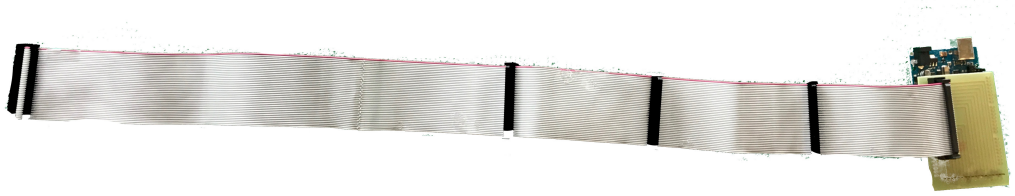

Figure 3.24: Flat cable with shield