Although any build system can be used with Qt, it brings its own qmake (that automates the generation of Makefiles). Even though 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 (as 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 [58]. It can be programmed by some special registers.

For the purpose which is going to be explained in Section 3.1.2, for this project the use of timers and their features was necessary.

The Arduino Mega 2560 CPU is the Atmel AVR ATmega2560. As it can be seen in its datasheet [59] 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 [60].

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 3.1.2, 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 this project, those libraries are *TimerOne* [61] and *Timer4* [62].

This libraries have the following functions [61]:

- $initialize(period)$: 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*().

In this project the most used functions were *initialize*() and *attachInterrupt*(). The minimum period or highest frequency this library supports is 1μs or 1MHz.

2.3 Components Integration

Figure 2.10 shows how the components that were previously mentioned in this chapter integrate the solution. The user can control via computer a graphical interface designed with Qt. Using ROS, the information given to the interface by the user is sent to the Arduino that proceeds to activate the vibration motors. The effect of the vibration motors is then felt by the user.

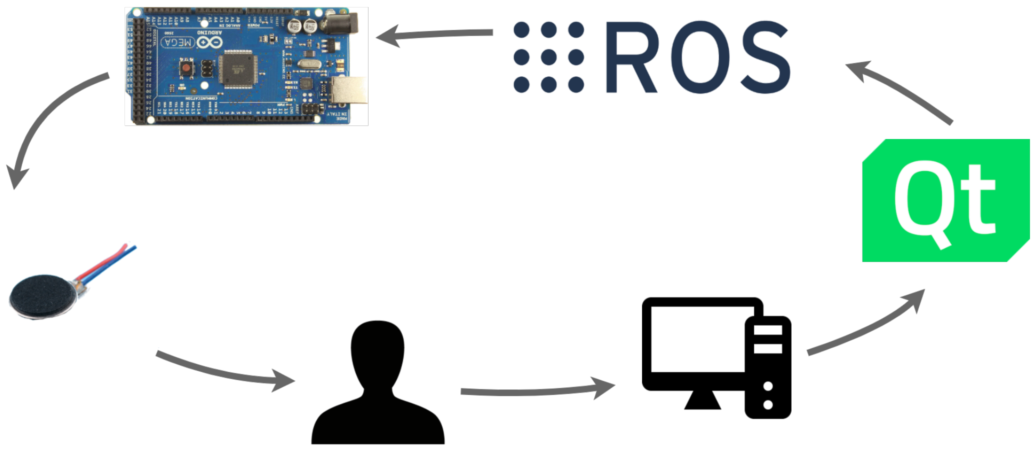

Figure 2.10: Diagram of all the components

Chapter 3

Development of an Application

This chapter describes all 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 Motor Control

3.1.1 ROS nodes

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

Figure 3.1 shows a flowchart of the three nodes that where created to achieve that goal.

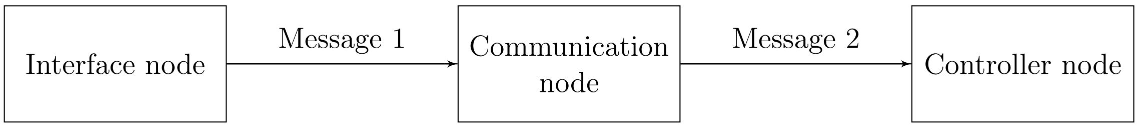

Figure 3.1: ROS nodes flowchart

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). In this node the user can interact with the computer through a graphical interface and choose which motor to turn on and its intensity.

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 activate te correct motors with the correct duty-cycle.

Figure 3.2 shows the ROS nodes layout made with the *rqt_graph* software.

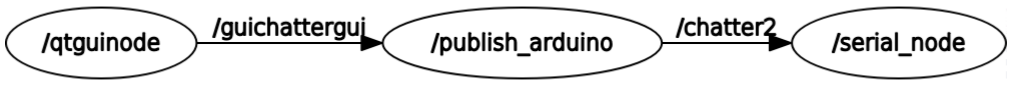

Figure 3.2: ROS nodes layout made with *rqt_graph*

3.1.2 PWM manually generated

As 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 signals 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 [60]. According to the Arduino Mega data-sheet, the priority of the different timers is different [59]. 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 3.3 is a scheme for an easier comprehension. More about this PWM generation will be explained in the following Section 3.1.3.

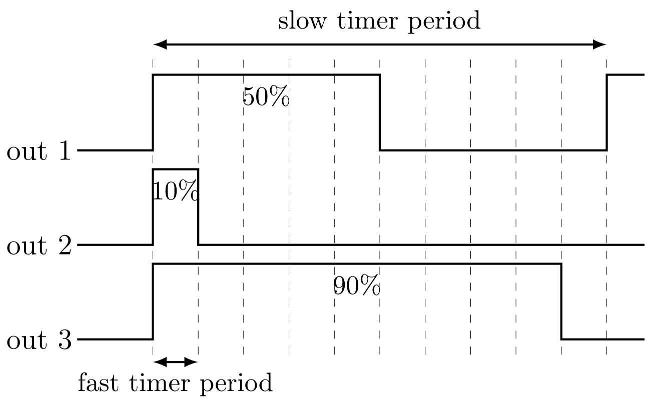

Figure 3.3: Timers behaviour

3.1.3 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 location of the motors.

When there is a set of objects (as the location of the motors) and a way to represent which objects to pick is needed there are two forms to do it:

- Map;
- Bitmask.

The Map associates with each object a boolean value, indicating whether the object is picked or not. However, this method takes up a lot of memory and it can be slow. Therefore a bitmask is more efficient [63].

A mask defines which bits to keep and which bits to clear and it is used for bitwise operations. Masking (the act of applying a mask to a value) can be accomplished by [64]:

- Bitwise ORing in order to set a subset of the bits in the value;
- Bitwise ANDing in order to extract a subset of the bits in the value;
- Bitwise XORing in order to toggle a subset of the bits in the value.

An integer is a group of bits stringed together. The first bit of this integer represents whether the first object is picked, the second bit represents whether the second object is picked and so on. For instance, if in a set of five objects the first, second and fourth objects are picked and the third and fifth are not, then the bitmask to represent this in binary is 01011 (or 11 in decimal) [63].

More about the creation of the bitmask will be explained in Section 3.4.1. For now, the mask that defines which motor to turn ON or OFF will be called motorMask.

Figure 3.4 shows the main structure of the controller program.

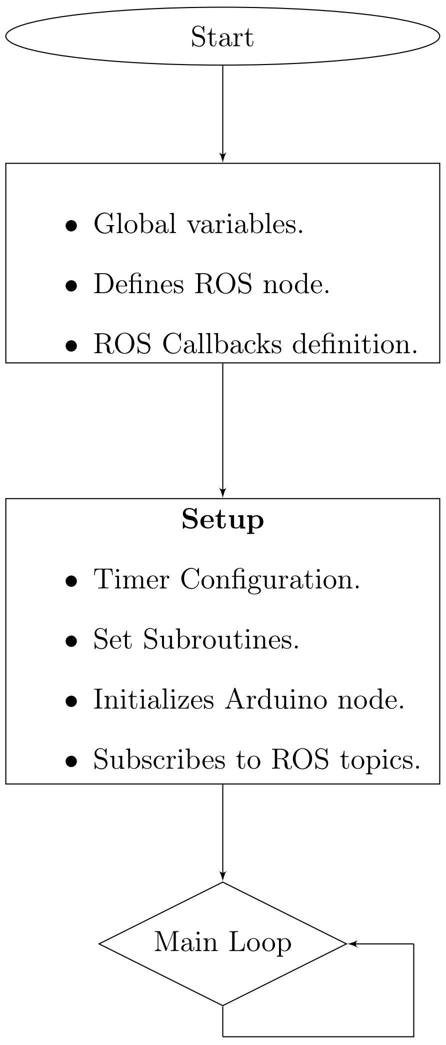

Figure 3.4: Controller flowchart

The data structure of a motor i can be define as following:

$$motor_i \begin{cases} motorMask_i \\ pwm_i \\ counter \end{cases}$$

In here, $motorMask_i$ is the bit value of the motorMask on position i, pwm_i is the reference duty-cycle value for motor i. As for *counter* it represents the number of times the fastest interrupt subroutine was called (meaning the current duty-cycle value).

For example, if the motor number five is the only motor supposed to be ON, it has a duty cycle of three and the controller just received this information (making the counter still equal to zero), it's data structure would be the following:

$$motor_5 \begin{cases} motorMask_5 = [0000000000100000]\ (binary)\ or\ 32\ (decimal) \\ pwm_5 = 3 \\ counter = 0 \end{cases}$$

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. Every time the timer 1 reaches its timeout, 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 3.5 illustrates the Interrupt Subroutine related to timer 1.

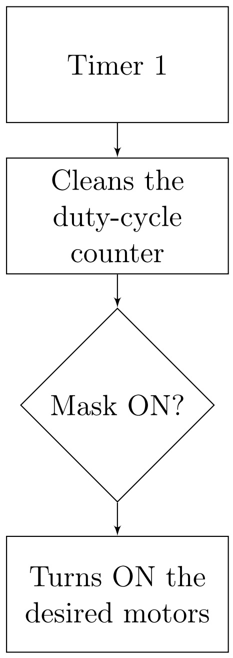

Figure 3.5: Scheme of the Interrupt Subroutine Timer 1

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 3.6 shows the ISR related to timer 4.

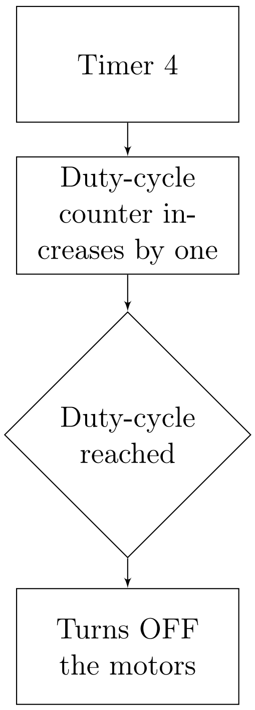

Figure 3.6: Interrupt Subroutine: Motors OFF

3.2 Peripheral nervous system and dermatomes

3.2.1 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 3.7, the dermatomes that the motors need to focus on are C5, C6, C7, C8 and T1.

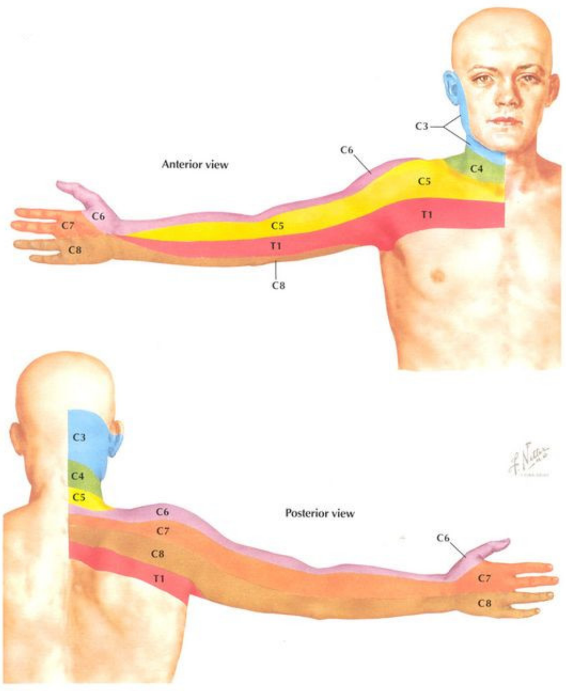

Figure 3.7: Schematic demarcation of dermatomes in upper body [34]

Considering this information, Figure 3.8 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.

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 3.1.1) 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.

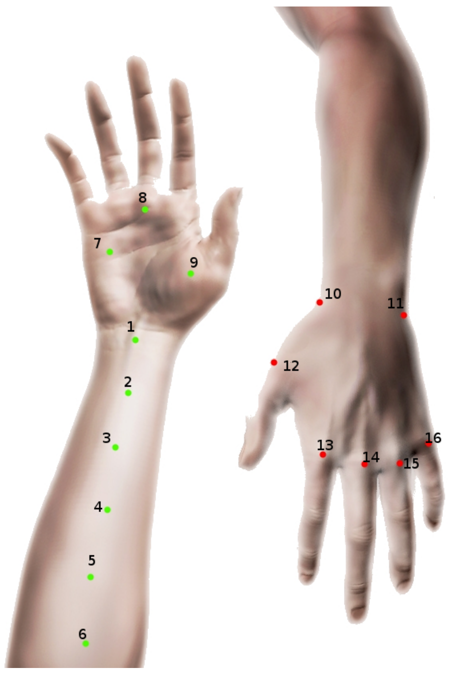

Figure 3.8: Placement of the motors (image adapted from [65])

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.

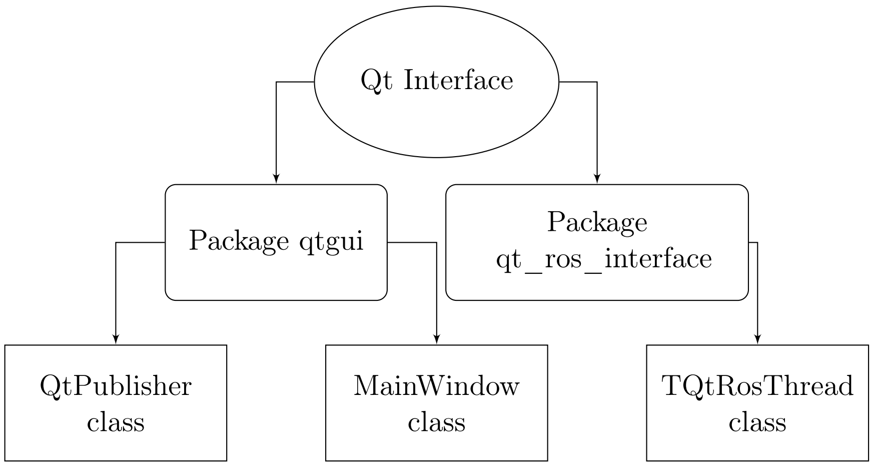

Figure 3.9: Interface organization

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.

```
qtgui
├─ msg
│  └─ GUIDados.msg
├─ src
│  ├─ main.cpp
│  ├─ mainwindow.cpp
│  ├─ mainwindow.ui
│  ├─ QtPublisher.cpp
│  └─ images
├─ include
│  ├─ globals.h
│  ├─ mainwindow.h
│  └─ QtPublisher.h
├─ CMakeLists.txt
└─ package.xml
```

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.

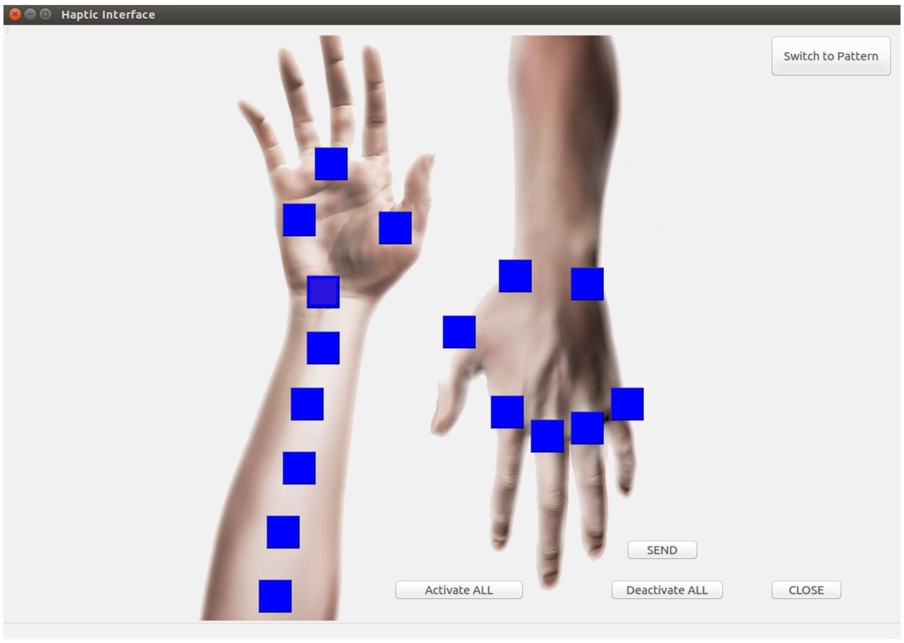

Figure 3.12: Interface: Motor Control

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:

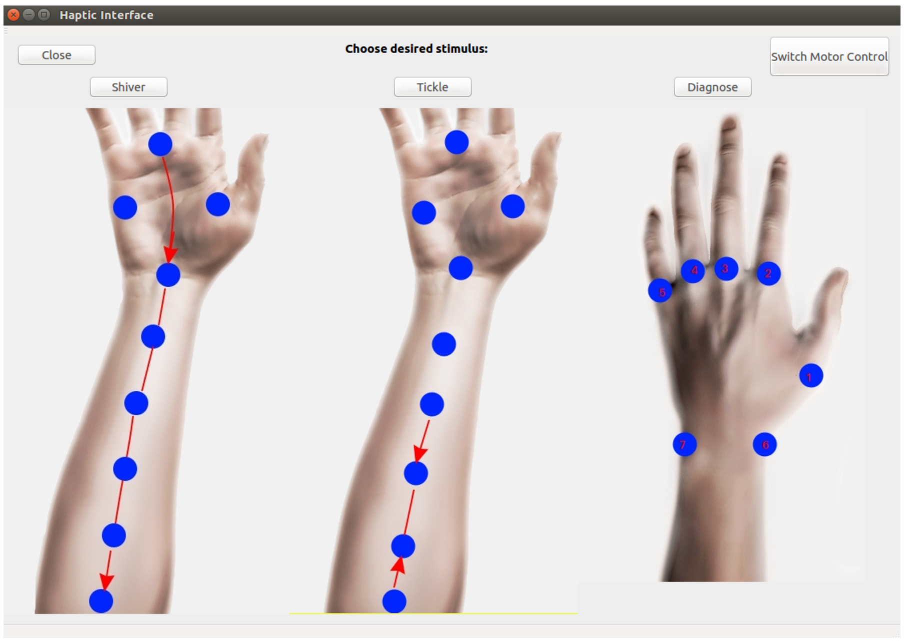

Figure 3.13: Interface: Pattern

- 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.