# POLITECNICO DI MILANO

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING

BIOMEDICAL ENGINEERING

POLITECNICO
MILANO 1863

**DESIGN AND REALISATION OF A TABLE-TOP ROBOTIC GAME FOR MOTOR IMPAIRED CHILDREN**

Advisor: Prof. Andrea Bonarini

Bachelor Thesis by:


| Fabio Paini | Beatrice Pazzucconi | Damiano Quadraro |
|---|---|---|
| ID 806874 | ID 809830 | ID 806809 |


Academic Year 2015 - 2016

# Contents

**List of Figures** **III**

**List of Tables** **III**

**Abstract** **1**

**Sommario** **2**

**1 Introduction** **3**
1.1 Social Robotics . . . 3

**2 Related Studies** **5**

**3 Concept** **7**
3.1 Focus on the problem . . . 7

**4 Methods** **9**
4.1 Creation of Game/Rules . . . 9
4.1.1 The Story . . . 9
4.1.2 The Game – FlipperBot . . . 9
4.2 Controller . . . 10
4.2.1 Joypad . . . 10
4.2.2 Accelerometer Wrist/Head Band . . . 11
4.2.3 Joystick . . . 11
4.2.4 Virtual Interface . . . 13
4.3 Robot . . . 14
4.3.1 Frame . . . 16
4.3.2 Electronics . . . 17
4.3.3 Head . . . 17
4.3.4 Interaction and Feedback . . . 18
4.4 Board . . . 18
4.4.1 Display . . . 20
4.4.2 LEDs . . . 21
4.4.3 The “everything button” . . . 21
4.5 Totems . . . 21
4.6 Software . . . 24
4.6.1 Scheduler Module (ScheMo) . . . 25
4.6.2 Network topology . . . 28
4.6.3 FlipperBot Communication Protocol (FBCP) . . . 29
4.6.4 Joystick . . . 31
4.6.5 Robot . . . 31
4.6.6 Board . . . 32
4.6.7 Virtual Interface . . . 35
4.6.8 Motor speed analyser . . . 35

**5 Method Evaluation** **39**
5.1 Proposed tests . . . 39
5.1.1 Tests During Realisation . . . 39
5.1.2 Tests After Realisation . . . 40
5.2 Performed tests . . . 40

**6 Results** **42**

**7 Conclusions** **44**

**8 Future Developments** **45**
8.1 Possible Improvements . . . . . 45

**9 Acknowledgements** **47**

**Appendices** **48**

**A Circuits and Electric Schemes** **50**

**B External Resources** **51**
B.1 Raspberry Pi (RPi) . . . . . 51
B.2 Arduino . . . . . 52
B.3 ESP . . . . . 52
B.4 Themes and Sounds Effects . . . . . 53

**C `schemop` directives** **54**

**D FBCP commands** **58**

**E External display messages** **63**

**F Code Snippets** **65**
F.1 C++ . . . . . 65
F.2 Python . . . . . 70

**Bibliography** **73**

# List of Figures


1.1 Examples of Social Robots. . . . . 4

2.1 Examples of social robots used with disabled children. . . . . 5

4.1 3D render of the joypad . . . . . 11
4.2 Realised Joystick. . . . . 12
4.3 Virtual joystick interface . . . . . 14
4.4 Realised Robot. . . . . 15
4.5 CAD models of the inner structure of the robot. . . . . 16
4.6 Electric scheme of the robot's control board . . . . . 17
4.7 Realised board with totems. . . . . 19
4.8 CAD of assembled board with part of the cover removed. . . . . 21
4.9 Realised Totems. . . . . 22
4.10 CAD model of the structure of totems. . . . . 22
4.11 Detail of realised totem base and mating. . . . . 23
4.13 Detail of the magnetic mating. . . . . 24
4.14 Structure of the main software components of this project . . . . . 25
4.15 Example flow diagram of a ScheMo program. . . . . 26
4.16 Network topologies . . . . . 28
4.17 `Monitor` interface . . . . . 33
4.18 `demo` interface . . . . . 34
4.19 Graphical representation of the configuration options . . . . . 37
4.20 Steps of the analysis of a frame . . . . . 37

6.1 Boxplots of acquired data . . . . . 42
6.2 Boxplots of acquired data (continued) . . . . . 43

7.1 Plot of the periods of rotation for each sample. . . . . 44

A.1 Scheme of Joystick Circuit. . . . . 50
A.2 Scheme of Totem Circuit. . . . . 50

E.1 Formal grammar definition of display messages . . . . . 63

F.1 Partial flow diagram of ESPer's program . . . . . 72


# List of Tables


C.1 List of ScheMo Preprocessor directives . . . . . 57

D.1 List of supported FBCP commands . . . . . 61
D.2 List of standard FBCP parameters . . . . . 62

E.2 List of displayable symbols . . . . . 64

# Abstract

Since 1950's playing has been regarded as crucial to proper child's social, personal and physical development. This applies not only to games specifically designed to educate, but also to all forms of playing, that is: "play for the sake of play".

Children with disabilities in particular seem to react in an extremely positive way, although they're often secluded from normal children's activities, and therefore are deprived of a valuable opportunity to develop and overcome some of their limitations.

Our work focused on creating a game environment for children with motor impairments and a table-top interface to play it. The game provides kid-friendly interactions and the possibility to play with another child (whether disabled or not), trying to even the participants abilities.

In this relation we explain the various steps taken to design shape and interaction of the game, prototype construction, in prevision of an eventual trial with patients. Although it may sound obvious, we would like to point out that both the design and construction were entirely carried out by ourselves and almost exclusively with our own means. Therefore, also owing to the little time or poor resources available, we sometimes resolved to solutions different from what we initially planned. This relation separates the design process from realisation, so that it is easy to understand where we had to come to a compromise or alternative solution. Also, options are compared in a future development-perspective, for whoever might like to improve our results.

# Sommario

A partire da circa la metà del secolo scorso, si è cominciato a prestare un crescente interesse nei confronti della psicologia dei bambini. Ne è emerso che il gioco rappresenta una parte fondamentale per lo sviluppo corretto del bambino, sotto molteplici punti di vista: fisico, psicologico e sociale. Un aspetto importante di questa scoperta è inoltre che il gioco non necessita di avere uno scopo prettamente educativo per sortire il suo effetto positivo, ma può anche essere fino a se stesso.

I bambini con disabilità beneficiano più degli altri di questi effetti, ma paradossalmente, vengono esclusi da questa possibilità spesso, perdendo così l'opportunità di superare le loro difficoltà.

Il nostro progetto si propone di creare un sistema di gioco da tavolo ed un'interfaccia di gioco robotica per bambini affetti da disabilità motorie. Il gioco promuove una comunicazione costruttiva nei confronti dell'utente e fornisce l'opportunità di avere un secondo giocatore (normodotato o meno), cercando al contempo di portare i due giocatori allo stesso livello.

In questa relazione illustriamo i diversi passaggi effettuati per progettare la forma e l'interfaccia del gioco, la costruzione di un prototipo, in vista di un eventuale test con pazienti.

# 1. Introduction

By now there is strong evidence in literature that play is not only a purely leisure activity, but that is fundamental to allow children develop properly. Proof of it is the decision of the U.N.O.[1] to make play a right of every child.
Playing:

- develops social, physical and cognitive abilities;
- improves psychological and emotional strength.

What is even more interesting is that it is not necessary for the game to have a specific educational target, but also playing for its own sake serves the purpose. Moreover, there is evidence that deprivation of playing may affect negatively the child in the future.

Despite this, many children are still denied this opportunity, and this can be due to:

- living conditions (poverty, wars, exploitation...);
- progressive emphasis on academics in schools and highly scheduled lifestyle trends in middle and upper class families [3];
- children disabilities.

Consequences of this in adulthood and adolescence may be: (only relating to the second and third classes)

- anxiety, depression and lack of self-confidence and self-esteem in general [3];
- lowered motivation, dependence on others, underdeveloped social skills with particular regards to disabled children [10].

Children affected by disabilities seem more prone than the others to suffer the consequences of deprivation of play and proper stimuli. We can define a "physical" deprivation (lack of sensory and motor information received by the child), and a "social" deprivation (secondary disabilities as an indirect result of the former). The reasons for this are various, such as physical limitations of the child him/herself, environmental barriers, and social barriers (integration with peers) [2],[3],[9],[10].

Recently, many efforts have been made to use technology to assist disabled children. Also a 4-year action supported by the C.O.S.T.[2] started in 2014 to connect experts and professionals all over Europe (from 28 countries) and spread concern, to ensure children with disabilities are granted their righteous playing. (LUDI — Play for Children with Disabilities[3]) [5].

## 1.1 Social Robotics

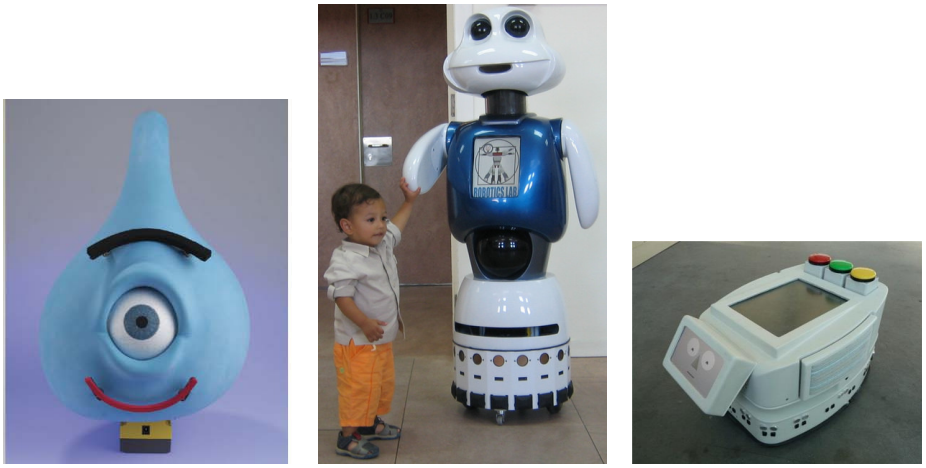
Robotics seem particularly promising in this case and have been already employed quite successfully in biomedical environment. In particular it is Social Robotics' aim to develop a natural and productive human-machine interface to improve the quality of playing.

Some encouraging experiments have already been carried out in favour of child-robot interface rather than a bare interaction with a tablet or device (for example to measure children's ability to develop language).

[1] United Nations Organization

[2] European Cooperation in Science and Technology

[3] See `http://www.cost.eu/COST_Actions/TDP/Actions/TD1309` and `http://www.ludi-network.eu`



**(a)** Emuu: Bartneck, C., Reichenbach, J., and Breemen, A., 2004

**(b)** Maggie: Miguel A. Salichs, et al., 2006

**(c)** Van den Heuvel R. et al., 2015

**Figure 1.1:** Examples of Social Robots.

In fact, children generally react to the robot with more ease as with an adult, for it doesn't look as inquiring or expectant, as long as the robot is fitted with an opportune appearance (that is, it needs to be dressed up to look child–friendly). Therefore children react more enthusiastically to them as they would with simply a tablet [13],[14].

# 2. Related Studies

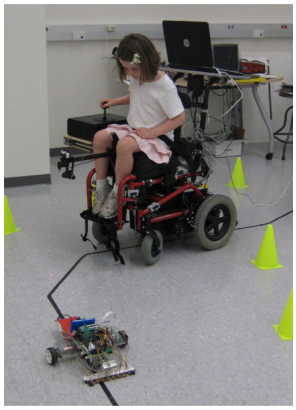
Social Robotics potential has been evidenced also in experiments/studies involving disabled children. Patients were mainly children affected by ASD[1], motor and generally cognitive disabilities [1],[7],[8],[11], or even diabetes [6], and much more is still in progress [12].

**(a)** LEKBOT: Ljunglöf P. et al., 2011

**(b)** Westlund J. et al., 2015

**(c)** Charlie: J. Cabibihan et al., 2013

**(d)** KASPAR: Dautenhahn K. et al., 2009



**(e)** Marchal-Crespo L. et al., 2010

**Figure 2.1:** Examples of social robots used with disabled children.

Difficulties encountered by researchers were mainly:

1. low playfulness of patients (whether due to physical impairments or to the child's mood);
2. wide range of possible impairments; that is, every patient, especially those affected by ASD or motor disabilities, has his/her own peculiar situation.

The main limitations of the above–mentioned studies (and in general, studies related to the use of social robotics in children playing) were:

1. almost exclusive focus over children with ASD, neglecting other disabilities;
2. excessive focus on the educational purpose of the game;

[1] Autistic Spectrum Disorders

3. lack of possibility to play with other children and therefore improve social skills;
4. excessively complex/simplified game schemes and rules so that the game may be regarded as too hard/boring;
5. excessive/too apparent facilitation of disabled child that undermines challenge.

# 3. Concept

Playing is good for children, under many aspects of their development (cognitive, relational, psychological, social, physical...). Although disabled children are deprived of this possibility more often than the others, it seems that playing would even be more productive for them. This project aims to verify this assumption, with particular regards to the social benefits of play for motor disabled children.

We focused on children with motor disabilities, although some pathologies typically affecting neuromuscular system may also result in sensitive and cognitive issues, e.g. quadriplegia, cerebellar ataxia, hemiparesis... In fact, up to now, the main addressees of the experiments were children with ASD, and therefore the games were designed for children who could move (in most cases) with little physical restrictions.

Little in comparison has been made for children with motor disabilities, and the employment of robotics up to now hasn't produced the same results as with children with ASD.
This has 4 main reasons:

1. The range and severity of difficulties in movements is very wide even among patients with the same pathology, and also because of the variety of different illnesses that can result in motor disabilities
2. Often, as the game–rules are extremely simplified to allow children to succeed, the game gets boring and repetitive very quickly. On the other hand, a game should not be too complicated so that the child can play freely
3. Low playfulness of patients (due to the difficult interface with the robot but especially due to the impossibility of playing with peers)
4. Complicated and time–expensive interface needed in the most severe cases (such as quadriplegia) e.g. Eye Tracking Devices, Sip-and-Puff switches, head wands or mouth sticks, Voice Recognition software...

## 3.1 Focus on the problem

Since time was short to design interfaces able to deal with all of possible impairments, we needed first to select a specific target of patients.

**Inclusion Criteria:** we chose to work for children with medium motor impairments, and which do not have severe cognitive and sensitive problems (reserving the option to adapt the interface for other types of conditions, such as tetraplegic patients, at least in a future development–perspective).

**Age:** 7–12 years (or mentally equal in case of cognitive disabilities).

**Limitations:**

1. the child is possibly sat on a wheelchair and may not be able to move much further from it;
2. his/her movements may be imprecise and/or sudden, he/she may not be able to calculate strength;
3. child is possibly not able to speak properly and may need time to plan the movement as he/she may not hear/see perfectly.

**Problems to be solved:**

1. the game must not be boring after a few times,
2. at the same time, the game must be simple enough for the child to play even with difficulty in movements;

3. multi–player possibility: the child must be enabled to play with others (whether other disabled children or not) without the game looking fake (too much advantage for the patient) or impossible to win. That is, the game should provide a way to even the possibilities of each;
4. the robot must be safe for the child to handle if he/she wishes to, and possibly be satisfactory on the sensory point of view;
5. the robot should possibly give the child feedback when playing, just as if it was a peer itself.

# 4. Methods

## 4.1 Creation of Game/Rules

### 4.1.1 The Story

ESPer is a little owl–shaped alien, employed at the PIAP Inc. (Pluto is a Planet), a company based on Volcano, in a not so far galaxy. His job is to keep an eye on some old satellites in the orbit of the planet Dabón. These satellites are very old and sometimes their program stops working. The way to fix them is to crash ESPer's pod onto their panels.

The network of these satellites is owned by Sirocco Studios, that creates cartoons for children, and therefore it is very important to keep it active. When the satellite is lighted up it means it is broken and needs fixing. If the satellite does not start working again in a few seconds, the whole network crashes and needs a whole day to be brought back to normality. ESPer is short–sighted and needs help to do his job.

Help ESPer keep children happy!

### 4.1.2 The Game – FlipperBot

We took inspiration from pinball and bumper cars to create a game in which the goal is to drive a small vehicle onto a board in order to crash into totems that are light up randomly.

The vehicle (from now referred to as Robot) is controlled by the child with a very simple joystick, while the lighting of the totem is controlled by circuitry inside the board. The board also provides a display onto which the score and time is showed, and is fit with some barriers on the borders that prevent the robot from falling out. Both the robot and the board are fit with sound/light effects to make them appealing.

**Single Player Mode:** every set time, the board would decide to light up one of the five totems, and the child should try to hit it as quickly as possible. If he/she succeeds before time is over, he/she scores.

The time given to complete the task is 13 seconds for the first 15 points, then 10 for the following 15 points, then 7 seconds for the last 15 points. Is the user manages to hit all of the totems, then he/she wins the game.

**Two Players Mode:** two children may play one against the other to reach the higher score by hitting totems more quickly (due to little time available for design and construction, we focused only on single player mode).

In order to even the abilities of the two players, it would be interesting to program robots to arbitrarily not execute the command from players but to go randomly around, as if it was their choice. More details about this can be find in section 4.3.

**Simon Says Mode:** playing mode inspired by the popular children's play. It can be suitable for younger children, or if robot batteries are recharging, for it relies on board and joystick only. When the board sees that only the joystick is connected, it asks the user if he/she wishes to enter this mode. The user must confirm he/she wants to enter Simon Says mode by pushing down the joystick, or wait for the robot to connect.

The game starts with a set of ten lives. The board will tell the user when the sequence is given and when is it his/her turn to repeat it. The sequence is showed lighting up randomly four totems (the one in the middle is excluded). The user must then replicate exactly the sequence shown by tilting the joystick to light up the totems. Each time the user fails, he/she loses one life. The last sequence is then re-proposed. The game ends when no lives are left.

The length of the sequence is determined randomly through the following formula, depending on the difficulty.:

$$L_{max} = \frac{\text{Difficulty}}{5} + 1$$

$$L = \left\lceil \frac{1}{10} \sum_{i=1}^{10} rand\,(L_{max}) \right\rceil$$

Difficulty increases linearly with every success in a row (that is, if the user makes a mistake, then difficulty is reset to zero).

The scoring increases as the number of successes in a row: for example, if the user gets three sequences correct in a row, he/she will receive 1 point for the first, then 2 for the second, then 3 for the third. If the user makes a mistake, the score for the following success drops back to one.

## 4.2 Controller

The controller represents the most interactive part of the game, for it is the connection that allows the child to express his/herself. We developed a few ideas that should cover a wide range of possible impairments, but due to lack of time, we chose to develop only one of them.

### 4.2.1 Joypad

Controller with 3 very large buttons that order the robot to accelerate forward (middle button), or to spin to the left or right (left and right buttons).

The joypad may be set on the table with the board or secured to the wheelchair armrests using straps.

**Advantages:** this type of controller offers the possibility to patients affected by distonia or spasms to drive the robot without accidentally hitting the wrong buttons. It also can be decorated easily and provided with sounds or lights to make it more appealing, since it is quite large (approximately 50cm wide)

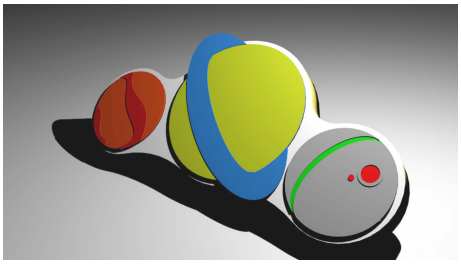


**Figure 4.1:** 3D render of the joypad

**Disadvantages:** not suitable if the patient is unable to reach the correct position to press the button (because of spasticity or paresis) or if is very weak, so that he/she tires easily. This controller was not developed because of lack of time and materials needed.

### 4.2.2 Accelerometer Wrist/Head Band

Wearable controller that can be tied to the child's wrist. The controller is simply an accelerometer and a control circuit that allows the patient to spin the robot on the left or right by swinging the arm (in this case the robot moves forward by itself).

**Advantages:** useful for patients who cannot reach the joypad/joystick or are too weak to push the buttons.

**Disadvantages:** may be still not useful if patient is tetraplegic or affected by spasm and sudden movements. Moreover, the use of accelerometers requires an accurate signal analysis. This is especially true when considering the filtering needed to adapt the interface to the movements of the patients, which can be imprecise, jittery or discontinuous.

Therefore, mainly because of lack of time, we chose not to develop such a controller, but it stands as a future development, in the perspective to create multiple interfaces that can cover the widest range of possible impairments.

In the case the patient is tetraplegic, the controller can be adapted to fit on the forehead too (just as if it was a head wand), but in this case the patient may get tired easily.

### 4.2.3 Joystick

Very weak children may not be able to swing/push the controller long enough to play effectively. If they have sufficient control over at least one upper limb, it is possible to create a joystick. It is sensitive enough to allow the patients to drive the robot without tiring, and has eight positions: forward, backward, left, right and intermediate between those. There should be also a ON/OFF switch and a slot for batteries. The joystick and its shell are smaller than the joypad and can be positioned where the patients can reach it without difficulty (using straps).

**Advantages:** can be fit onto wheelchair's armrests, just like normal controls of a scooter. These patients are very practiced in driving their scooters and therefore they should fare well with such an interface. It does not need strength.

**Disadvantages:** the patient needs control over one arm at least. Therefore it is not suitable for tetraplegic or distonic patients.

We chose to develop the joystick for it looked more immediate, since it resembles a well–known interface for wheelchair driving patients.

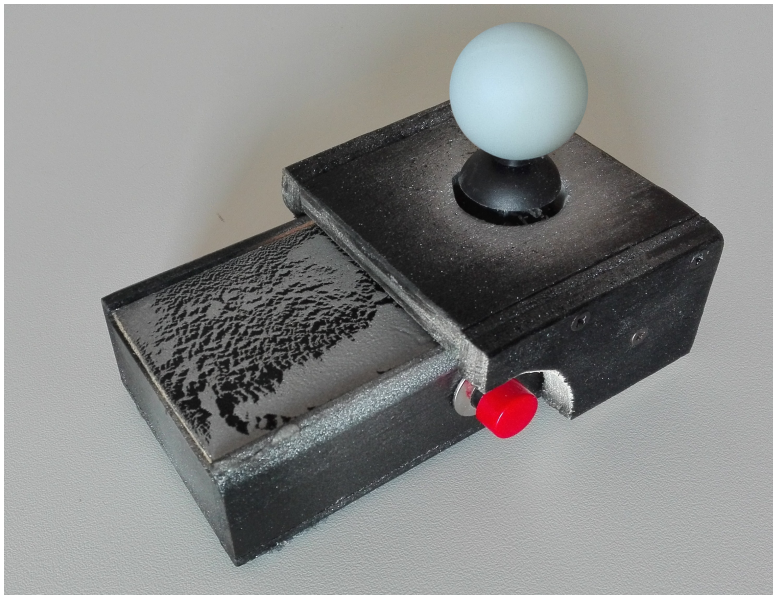


**Figure 4.2:** Realised Joystick.

### Mechanical part

The shell provides protection for electronics, support for the joystick and ON/OFF button, and access to battery slot. The shell was realised partly modifying a wooden box, and partly by building pieces ourselves. The shell should not be too large as it needs to be placed ideally besides the wheelchair's armrests, and should not hamper the child's movements.
The lid of the shell is made up of two parts:

- a sliding part that can be easily moved and allows to change batteries (which is the part the user should have access to);
- a fixed part that covers circuitry and provides support for the joystick, button and LED. This part can be removed by screws, so that control unit program or circuitry can be modified, but should not be accessible to children.

The joystick is secured to the fixed lid with screws.

### Electric part[1]

Requirements of this part are:

- provide power;
- interpretation of the inputs given by user (direction of the robot and pause);
- transfer data to board and receive from it (or from/to robot);
- give sensory feedback on the state of the controller;

The circuitry is powered by 3 regular 1.5V AA batteries, provided with their battery slot. We made this choice basing on the ease to find the batteries, size, and possibility to recharge them. Control unit (ESP-12) receives right input voltage from 3.3V regulator (AMS1117), that also provides power to the analog MUX

[1]For the electric scheme, see Appendix A

(PC74HC4053)[2]. The choice was made mainly on order to give the ESP-12 a regulated tension, that should remain stable for a while also when the batteries start running out. The OpAmp (LM378), on the other hand, is directly connected to the batteries.

Since the FSR[3] of the ADC on control unit is only 1V, a trimmer connected to the actual joystick reduces the voltage to a proper value. The joystick consists of two potentiometers (one for x axis and one for y axis), each a $5k\Omega$ resistor. The trimmer value is decoupled by a buffer, then given as input to the actual joystick (which is standard for Arduino interfaces). The outputs are given to the MUX, which is controlled by ESP-12 that selects inputs from the two axes alternately. Output is then given to ADC input of the ESP-12. This way we ensure a stable voltage supply for the potentiometers. The joystick also provides a button (which is activated by pushing the joystick down). This input is given directly to ESP-12 and serves as a "everything button", allowing the user to pause and resume the game, even if he/she cannot reach the "everything button" on the board.

The circuit is turned on by a spring button switch connected directly to the battery slot.

A red LED light signals when power is on, and a blue LED signals when the ESP-12 is online. The frequency of the blue LED flashing identifies the state of the controller at the moment:

**idle:** doing nothing: off;

**scanning:** searching for available networks: slowly blinking;

**connecting:** attempting connection with suitable server on the network: fast blinking;

**standard game mode:** connected to board and ready to play: alternatively fast and slow (this does not change when game is paused);

**standalone mode:** directly controlling the robot: no blinking.

#### Positioning

In order to place the joystick comfortably, we realised a strap (made out of Velcro), that can be tied to the wheelchairs armrest. A complementary Velcro is glued to the bottom of the joystick shell in order to ensure adjustment of the positioning.

There are two possible configurations, depending on the mobility of the child: the strap can be either wrapped around one armrest (for example, behind the scooter controls), or fastened across the space between the two armrests, so that the box can be placed in the middle. With this system, though sacrificing some stability, we ensure a wide range of positions that can be reached by the patient.

### 4.2.4 Virtual Interface[4]

In order to test and debug the game while work was still in progress, we created a graphic interface that can simulate both the totem system (with display) and the joystick. From this interface we created a simplified version of joystick that could be used to adapt the controller to different users. By creating an actual virtual controller, the child has the possibility to use his/her own tablet or laptop to play. Patients with most severe restrictions are often provided with personal computer interfaces (e.g. to drive wheelchairs) and a controller application would represent an immediate and flexible approach in order to broaden the range of suitable users.

[2]For more information about ESP-12, see Appendix B.3

[3]Full Scale Range

[4]For implementation information, see 4.6.7

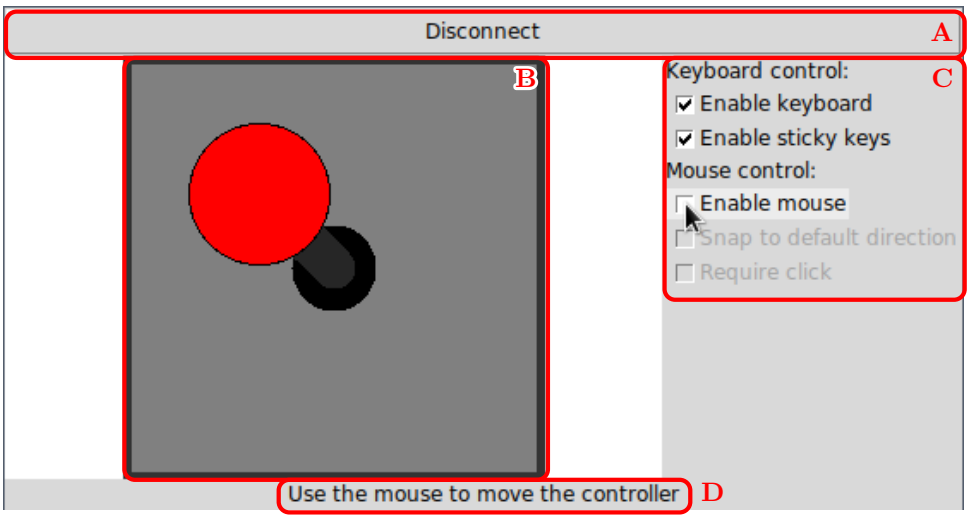



**Figure 4.3:** Virtual joystick interface
**A:** Connection button; **B:** Virtual joystick; **C:** Options; **D:** Hint text.

Figure 4.3 shows a screenshot of the interface. The labels refer to its components:

**Connection button:** allows to start and stop a connection with a board or a robot (the WiFi connection to the right network must be done outside of this program);

**Virtual joystick:** simulates a real joystick, showing its current position and accepting user input;

**Options:** allow to enable or disable certain features of the virtual interface on the fly, enhancing the user experience;

**Hint text:** briefly explains what is the meaning of an option when the cursor goes over it.

Depending on the selected options, it is possible to control the device either with a keyboard (arrow keys or number pad) or with the mouse (or with both at the same time). Regardless of the enabled input method for controlling the joystick, a virtual "everything button" is always bound to the `Return` keyboard key.

If the child commonly uses a hardware or software solution that simulates a keyboard or a mouse, it can be used with this interface too.

## 4.3 Robot

The second most interactive part of the game is represented by the robot (ESPer). If the controller allows the child freedom of expression, the robot is the part that deals with him/her as a peer[5].
It consists of three main parts:

- frame (that supports and protects the circuitry and shell), and wheels;
- electronics: batteries, regulation circuitry, lights, motors, control unit;
- outer shell (or "head") that can be decorated to form a personalised "outfit" and provides further protection for electronics.

[5]For implementation information, see 4.6.5

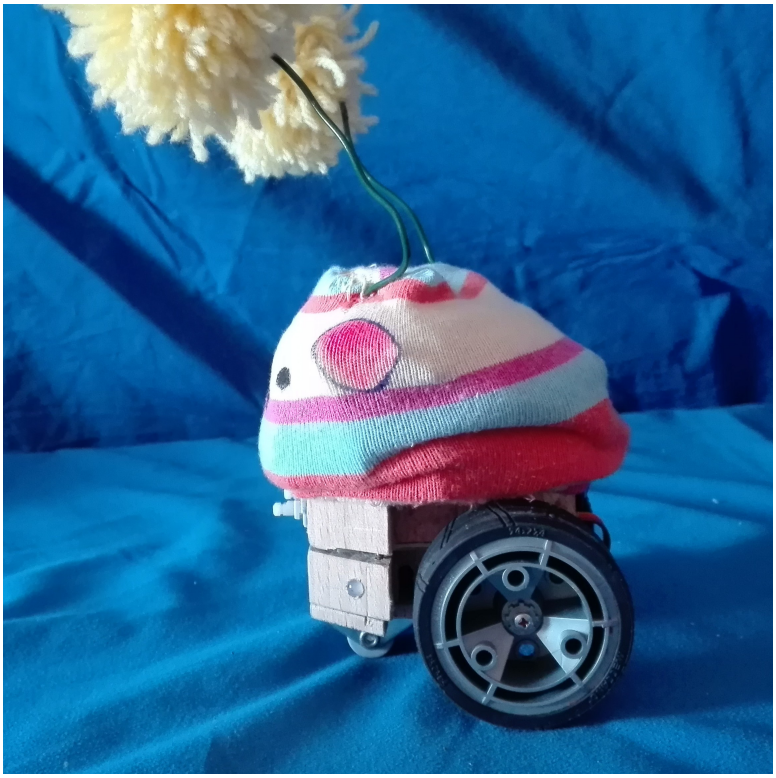


**Figure 4.4:** Realised Robot.

It receives instructions from the controller through WiFi connection. The wireless network developed allows the child to control the robot with the controller independently from the board: if the robot sees the board activated, it connects with it and the game can start (standard game mode), while if the board is not active (its network is not on), the robot can used alone (standalone mode). We chose to create also this second mode in order to allow the child to use the game even when the board is charging batteries, or if the board set-up is made difficult, due to its dimensions. In this mode the robot can be used on the floor as well.

The batteries are accessible by removing the outer shell from the top of the robot. To prevent the user from touching the circuitry inside, the batteries are stored in a separate section of the robot and only the connectors are accessible. Ideally, the best approach would be to have the batteries placed on the bottom and removable using their own slot, without having to remove the outer shell. This was not done due mainly to lack of material.

Moreover, the robot needs to be small enough to move freely on the board (approximately 7cm in diameter). The problem of the size is particularly pressing when approaching the corners of the board, for the space left between edges and totems is very narrow. This is due mainly to the size of the wheels. To reduce the effect of this, the robot is programmed to stop progressing while the bumper is pushed. That is: the robot ignores the command “forward” while it is pushing against and obstacle and only turns on the spot to avoid it.

### 4.3.1 Frame

The frame is made up of four layers (from top to bottom):

**Battery slot** ensures that the user's only contact with batteries is through the connectors pinned on the wood;

**Circuitry slot** accommodates all circuitry and it is connected above to batteries and below to motors. Also lodges rear LEDs, ON/OFF switch and bumper structure (micro switches with plaques on the outside);

**Motor slot** used to store the two motors, which are secured to the wood to ensure stability, and front LEDs;

**Bottom layer** serves as lid for the stack and as a base for a small spinning wheel that allows the robot to be balanced.

Sections are carved in lightweight wood and secured by screws, the shape is hexagonal, inscribed in a circle approximately 7cm in diameter. The mask is set on top of everything. The strict detachment of compartments is needed both to minimize the eventuality of accidental electric contacts and to increase stability.

The motors are actually servomotors that have been modified so that they can spin at 360°. This is due to lack of time, poor information supplied by datasheets on online markets, and cost of components. This arrangement causes the two motors not to spin at the same rate, so that it may be useful to verify that this problem does not affect manoeuvrability too much[6]. Anyway, considering the size of the board (70cm) and placement of totems, the robot needs not to move on straight line for more than 30cm, so that we can accept some deviation without the game to suffer from this.

On the other hand, an advantage of using servomotors is that speed con be selected directly from the control unit, with no need for a driver.

To simplify the game, we chose to set the wheels only to spin at one speed, both clockwise and counter-clockwise. Steering is achieved in two different ways, depending on the inputs from the controller (in this case, the joystick):

- by blocking one of the wheels, so that the robot turns while progressing (either forward or backwards). This corresponds to tilting the joystick to upper/lower left and to upper/lower right;
- by spinning the wheels in opposite directions, so that the robot turns on the spot. This corresponds to tilting the joystick to the left or right;

[6]See section 5.2 for more information

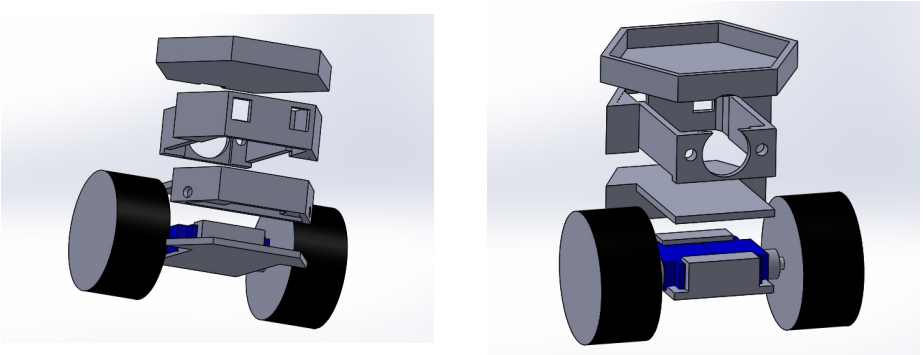


**Figure 4.5:** CAD models of the inner structure of the robot.

### 4.3.2 Electronics

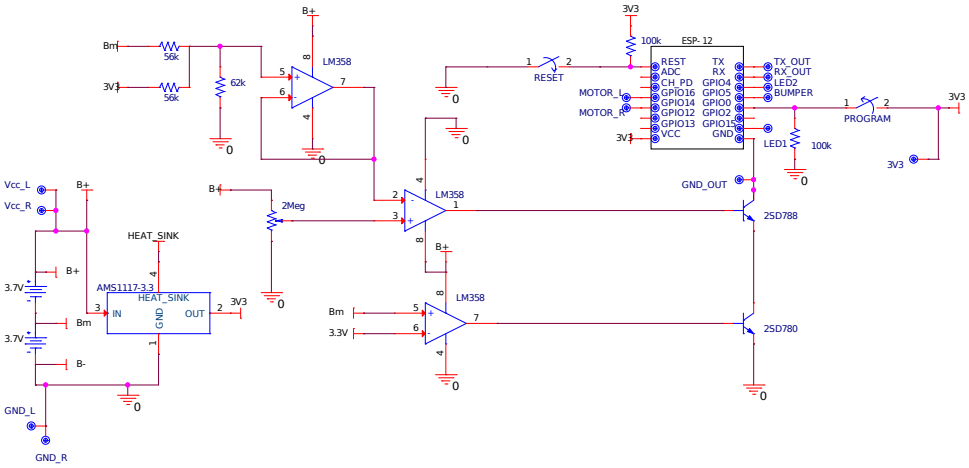



**Figure 4.6:** Electric scheme of the robot's control board

The robot is powered with two 4.7V LIPO batteries, and can be switched ON/OFF through a switch on the rear part. We chose to use LIPO batteries because, although more expensive, they are rechargeable and — compared to the alternatives (such as Stilo) — provide a better "power-to-bulk" ratio, for they are lighter and smaller than the others. The control unit (ESP-12[7]) requires 3.3V input voltage, given by the same regulator used in the controller (AMS1117).

The circuitry also provides a way of monitoring the voltage of the single battery through two triggers. When one of the two batteries drops below a threshold (set at 3.3V), the circuit cuts them off from the rest of the more power-consuming electronics (such as motors), preventing degeneration of batteries. The circuit provides connections between motors, LEDs, and bumpers, and ESP-12 as well as access for programming.

The circuit was designed entirely by us and printed as PCB by a Chinese manufacturer.

### 4.3.3 Head

The role of the outer shell is mainly decorative, but is it also useful to dampen impacts and protect the inner part. In this case, we gave ESPer a colourful look to appeal kids, and to ease realisation but also more "technological" outfits could be considered, such as space saucers etc. This is particularly true if we consider the possibility to play with another child, so that the two ESPers are recognisable.

The shell consists of a papier–mâché mask under a painted striped tissue cover fitted with an elastic band. There are also a couple of antennae that can be bent as one pleases fixed on the head. These were realised in metal wire and wool.

Ideally, the outer shell could also provide sensory feedback for the child, in case he/she wishes to handle it. This requires also the frame to be robust. Unfortunately, we had to neglect this side, due to lack of time/materials but especially owing to the need to continuously revise the circuitry and programming. Therefore the head is designed more to be easily removable (also because the batteries are accessible only from above) than to be sturdy.

[7]For more information about ESP-12, see Appendix B.3

### 4.3.4 Interaction and Feedback

Being responsible for an important part of interaction with the patient, it should have a range of sound/lighting effects.

- green and red LED lights (2 couples) that signal the state of the robot, like they are used in the controller:

  **idle:** green on, red off;

  **scanning:** searching for available networks: green on, red slowly blinking;

  **connecting:** attempting connection with suitable server on the network: green on, red quickly blinking;

  **standard game mode:** connected to board and ready to play: green and red alternately blinking (this does not change during the pause);

  **standalone mode:** directly controlling the robot: green and red both on.

- buzzer that emits sounds when the robot crashes, etc. We decided to delegate this part to the board as well, for the RPi can control the audio output more efficiently than the ESP-12 in the robot, providing more effective interaction, although sacrificing realism.

In a future development perspective, it our intention was to give the robot the possibility to disobey the user's commands from time to time, and randomly act on its own decision. This should not be as if openly helping the disabled child (in case the other player in not disabled).

It also should give the impression that the robots have actually their own will, but are not capable of doing thing on their own, so that the child perceives that he/she must help them (in the story, we justify this by saying that ESPer is short-sighted). We believe this is very important to develop self-confidence and social skills.

The sounds from the robot are provided by the board, and they are divided according to the situation in which they occur[8]:

1. entering the game from main menu;
2. resuming the game from pause menu;
3. scoring (hitting the totems);
4. running out of time (last 5 seconds of time allotted, when the totem starts flashing);
5. failure (not hitting the totem in time, or not hitting the right one in Simon Says mode).

## 4.4 Board

The board offers mechanical support the robot and totems, as well as central control unit and Wi-Fi network connection. Also, it provides lighting and sound effects to the whole interface and it is decorated accordingly to the theme.
It consists of three main layers:

- top layer, that can be open to operate on inner circuitry, and in which there are holes for the totems and for the removable boundaries. It is punctuated with white LEDs to mimic a starry sky, and it is covered by black plastic to simulate background;

[8] See also Appendix B.4

**Figure 4.7:** Realised board with totems.

- middle layer, that creates the room for inner circuitry and power supply and creates the connection between the sections;
- bottom layer, that provides basis for the totem mating and onto which middle layer is secured.

The display for scoring is located on the sides, as well as LEDs signalling the state of game, ON/OFF switch, and the "everything button". On the top there are also small holes that show the battery charge (being located above the battery pack).

The board was designed to be divided in four quarters that could be disassembled and transported more easily. The pieces are not the same size because of the middle totem: it would have been extremely difficult to create the basis for the mating if we had had to split it into fourths.

When assembled the body is 70x70cm wide and the thickness (mostly made up by middle layer) is about 5cm.

The sections are made out of lightweight wood and plastic and are connected with each other:

- mechanically (through wooden nails);
- electrically (through 4 channel 3.5mm jack plugs).

In order to ease the use of the game, all the circuitry and control unit (Raspberry Pi or RPi[9]) are located in the largest plaque (the one with two totems), from now referred to a "main section" .

The main section contains also a battery pack and a pin (micro USB) to access it. This ensures the RPi to receive its proper input voltage (5V) and that the board can be recharged without extracting batteries but simply by plugging in the main section.

Moreover, this way the entire game could be played with power provided directly from electric distribution (for the battery pack does not need to be disconnected to be recharged, unlike LIPO batteries).

[9]For more information about Raspberry Pi, see Appendix B.1

There is also another jack input to connect to game to an audio system (e.g. headphones, speakers...). The board on its own does not provide sounds if not connected to external device. Since it is lacking its own volume regulation, this allows the user to adjust it to his/her own preference, or even to exclude completely the audio.

The main section receives information from all other game components (robot and controller) via Wi-Fi, and from the totem via cables.
We chose Wi-Fi over other type of connections (e.g. Blue-tooth) for:

- ESPs and RPi provide built-in Wi-Fi connection;
- Wi-Fi connection is supported by a wide range of devices: laptops, tablets, cellphones... while others may not always be.

The drawback is that this kind of procedure is more energy consuming compared to Blue-tooth, for example. The RPi is responsible for collecting this information, processing it and sending it back.
These processes include:

- detecting the impacts between totems and robot;
- updating the score and reset when needed;
- lighting totems;
- counting time to the next target;
- transfer information from controller to robot;
- management of display;
- management of sound effects[10] (when connected to output).

The implementation needed in this process is extensive and so we describe in a separate chapter[11]. Battery pack, RPi and circuitry are secured into their positions by screws and metal bands. All pieces contain the basis for the mating with the totems (described in section 4.5).

The board also provides some boundaries to prevent the robot from falling off the top. They come in the form of wooden planks (5cm wide) with hinges that allow them to be folded and stored. They are secured to the top layer by wooden nails as well.

### 4.4.1 Display[12]

The display is a LTC4727G 7-segment common cathode, with four digits and seven points. It is controlled directly by Arduino UNO[13], that receives serial instructions from RPi. This is because, having many processes concurrently running, the Raspberry Pi does not manage to control the display most effectively, so that it tends to flicker quite visibly if directly controlling the display.

Apart from keeping track of the score and time countdown, the display is also used as a way to give more information about the game state and to improve appearance, contributing to the "space adventure" atmosphere.

For example, the display shows "FlipperBot" sliding from right to left in the main menu, or "LOSE" when time runs out, as well as some animations in between the various states.

[10] See Appendix B.4
[11] See 4.6.6
[12] For details on implementation, see 4.6.6
[13] For more information about Arduino, see Appendix B.2

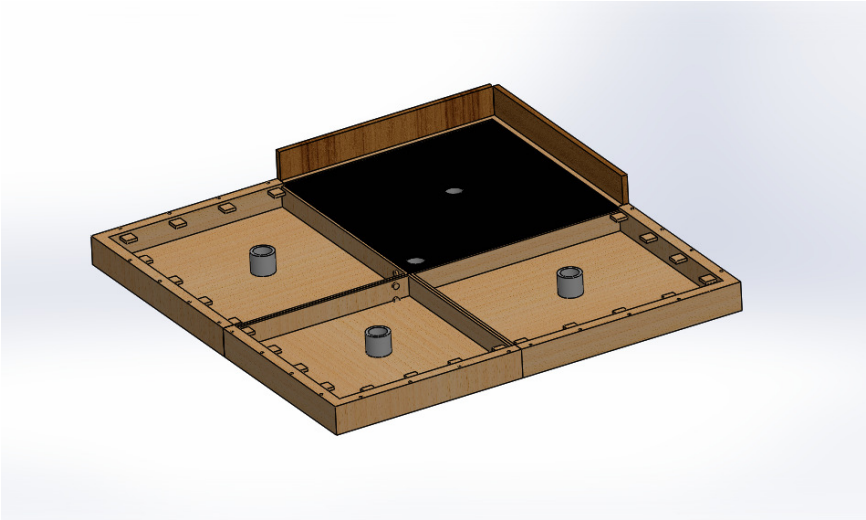


**Figure 4.8:** CAD of assembled board with part of the cover removed.

### 4.4.2 LEDs

There are three lights:

**ON/OFF LED:** red light signalling when the board is turned on;

**Player LEDs:** two yellow LEDs lights, one for each player. The state of game is represented by the frequency of flashing:

**LED is off:** both controller and robot are not;

**LED is on, no blinking:** both robot and controller are online, ready to play standard game mode;

**LED slowly blinking:** robot is online, but not controller;

**LED blinking fast:** controller is online, but not the robot.

### 4.4.3 The "everything button"

The big round button besides the ON/OFF switch allows the child to:

1. start the game by pushing once from the main menu (that starts automatically when turned on and to which the player returns in case of loss);
2. pause the game by pushing once from the game session. This stops the countdown and disconnects the robot;
3. resume the game by pushing once from pause menu;
4. return to main menu by holding the button 3s at least during pause.

All of these passages are supported by the change in music themes[14].

## 4.5 Totems

Totems are the targets into which the robot must crash in order to score.

They consist of cylinders containing the light set and switches (to detect impacts through plaques). The outside of the cylinder be decorated accordingly to the theme

[14]See Appendix B.4 for more information

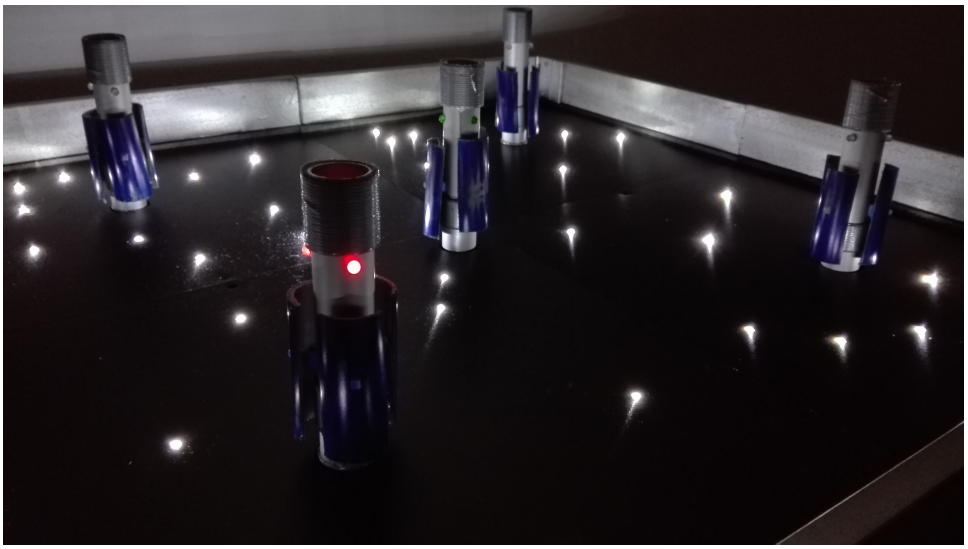


**Figure 4.9:** Realised Totems.

(e.g. as a satellite as we chose). In order to move the game more comfortably they are designed to be easily removable from the board.

### Mechanical Part

The core consists in aluminium cylinders (inner diameter 2cm). On the lower part of the core, there are three tough plastic plaques. The plaques transfer impact to the switches to which there are connected.

On the upper rim of the core three LEDs are positioned in the upper half of the structure and provide colour and feedback (two sets are blue, two are red and one is green).

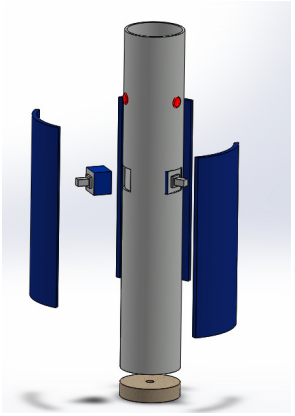


**Figure 4.10:** CAD model of the structure of totems.

### Electric Part

LEDs sets signal the totem that is active in that moment, and blink when the target is hit. The target blinks faster and faster as time left runs out. In the menu mode, totems are lit in a circle, waiting for the user to push the "everything button".

Lighting is controlled by the board onto which they are plugged, so totems do not need complex circuitry. The electronic part includes the contacts for the switches and LEDs (which are lighted up by RPi through a simple pull-up resistor) and power supply[15].

The choice to defer the control of the lighting to the board is preferable for:

- there is little space to work with inside the totem;
- allows saving of materials;
- allows to easily modify the connection if needed by operating only on the main section (for the board collects everything there).

[15]For the electrical scheme, see Appendix A

### Mating

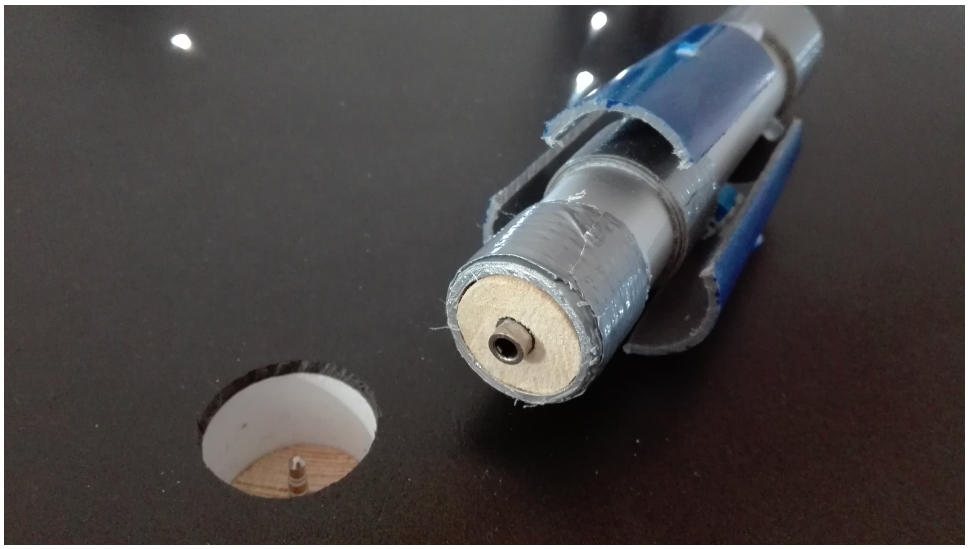


**Figure 4.11:** Detail of realised totem base and mating.

The mating provides ease of use, mechanical stability and electronically safe connecions. These are ensured by using the same 4 channel 3.5mm jack plugs as used to connect the board sections. They are stiff enough to ensure the totems do not tilt when hit by the robot. Moreover, they are quite a common component, so it is easy to find and replace them if needed.

Further stability is provided by a cylinder of tough plastic secured onto the bottom of the board, that prevents tilting and accidental contact among other wires in the board section. The complimentary plug is glued to a wooden disc from which the wires are gathered to be sent though connections to the main section.

In a future development perspective, it would be interesting to develop a mating system that should allow the child to insert the totems him/herself. We designed an alternative based on magnets: two complementary plaques would be fixed to the totems and board bottom. They would contain:

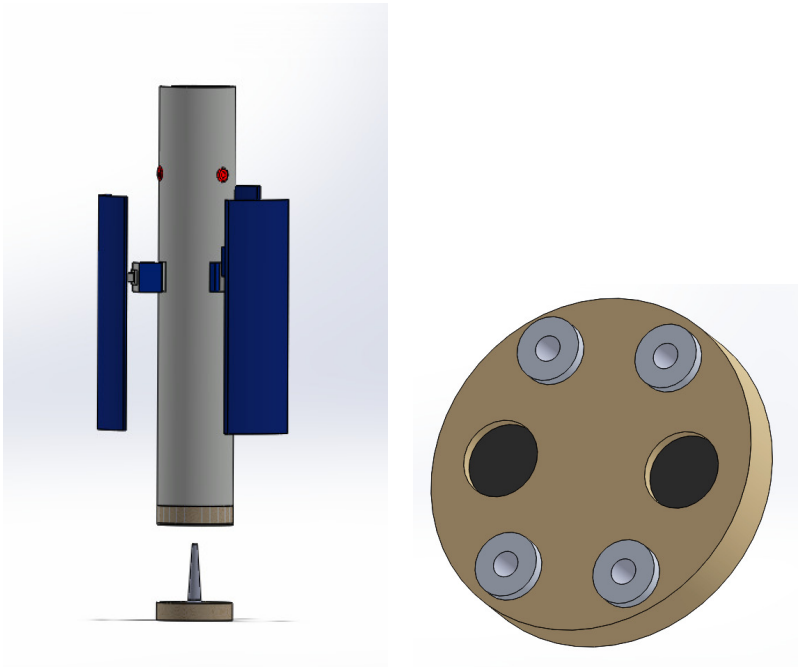
- **Magnets**: 2 small button magnets (6mm diameter) with opposite polarity. When totem is connected, only appropriate orientation (corresponding to the complementary magnets on the board) would allowed;

- **Electric contacts**: The connection would be provided by small screws and flat nuts for two contacts, while for the other two contacts it would be possible the magnets themselves.

On the board the corresponding plaque would be secured to the bottom layer. The plaque is slightly raised so that wires from the contacts can be gathered together and transferred to the pins, as with the current solution.

We tried to realise one of this plaques, but we had significant contact losses, especially when the totem was hit, for it tended to tilt and rotate. This is due to the facts that the magnets were not perfectly in contact, and also to the not exact alignment of the nuts. This is mainly due to the availability of technical means we had, since this mating requires such a precision we cannot reach with manual construction alone.

On the other hand, this development is potentially very interesting, since the child is provided with the possibility to assemble part of the game him/herself, and guided



**(a)** Model of totems with mating

**(b)** Detail of the magnetic contact plaque.

in a way that does not seem to facilitate him/her excessively (for the intervention of magnets is less apparent than, say, guiding lines traced on the core that point to the right orientation). This is very useful to give the child self confidence in his/her manual ability.

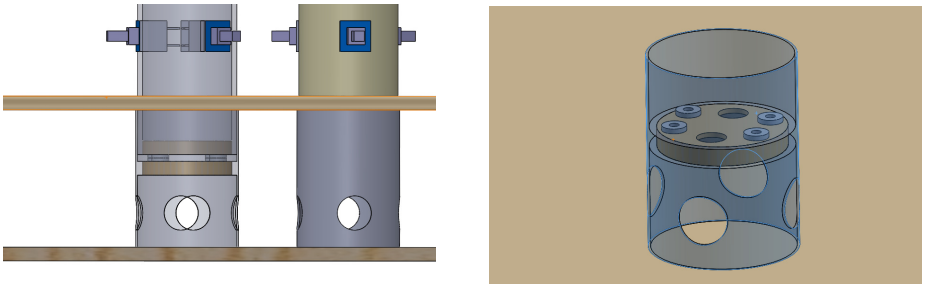


**Figure 4.13:** Detail of the magnetic mating.

## 4.6 Software

In addition to designing and building the physical parts of this project, a considerable amount of work was put in developing the software needed to make the various devices operate as wanted.

The main programming languages used were C++ and Python, with some utilities written as Bash scripts.

Figure 4.14 shows the main software components of this project, ignoring the utility tools only used during development.

Snippets of the code contained in this project can be found in Appendix F, presented as examples of use of the various tools.

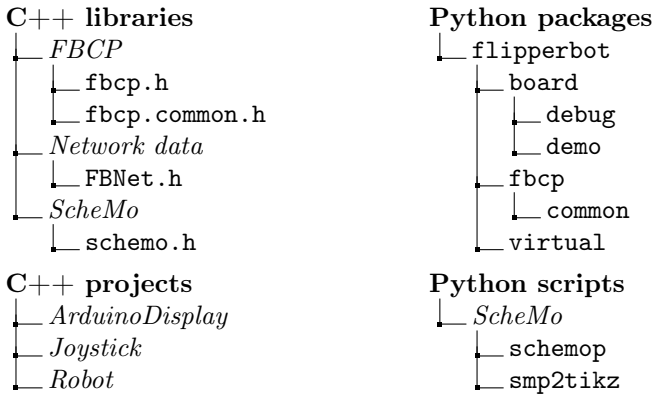



**Figure 4.14:** Structure of the main software components of this project

### 4.6.1 Scheduler Module (ScheMo)

Both the robot and the controller (as will be clear later) need to execute multiple tasks simultaneously to work properly, however the ESP-12 includes only a one-core CPU, allowing for only one process to run at a time.

The SDK[16] available to be used with the Arduino IDE[17] implements a simple scheduler to allow the correct functioning of the WiFi communication while the main program runs. Unfortunately, the related API[18] are hard to use, badly documented and their use is discouraged by the developers.

For these reasons, we developed our own scheduler, written in standard C++ and thus compatible with various platforms other than the ESP-12. It is, however, only a barebone implementation useful for this specific application but lacking of various features common to modern schedulers.

#### Control flow

To better understand how ScheMo simulates parallel processing, we first have to define some of its terminology[19]:

**Job** it represents a single (pseudo)process that can be started, stopped, paused and resumed at will. Different processes are represented by different jobs.

**Task** it is a continous, non-interruptable part of a job; that is, the scheduler can switch control from a job to another only after a task of the first job has terminated and before the next task starts. This means that it is vital to split jobs in as many tasks as possible and to make sure that tasks always halt (this condition is not required for jobs).

**Function** it is similar to a job in that it can run concurrently with other processes, but it has some important differences:

- it can't be scheduled by itself, it must instead be called inside a job (that passes control to the function until completion[20]);

[16] Software Development Kit

[17] Available at `https://github.com/esp8266/Arduino`

[18] Application Programming Interface

[19] Note that these definitions are specific for ScheMo and do not necessarily correspond to the meanings that the same words have in other contexts of Computer Science

[20] Asynchronous function calling has not been implemented yet

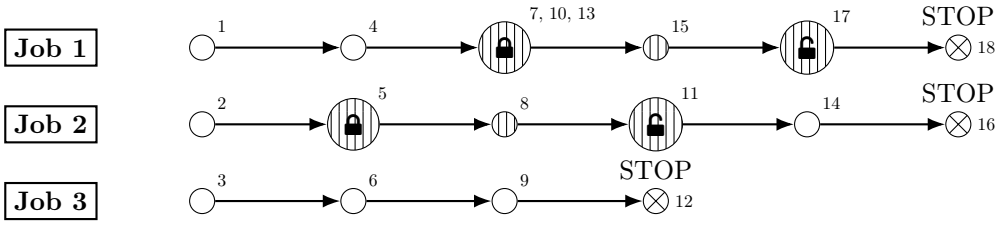



**Figure 4.15:** Example flow diagram of a ScheMo program.
Each job has a distinct flow diagram, where tasks are represented by circles.

The padlocks indicate if a mutex is locked/unlocked inside a task and they delimit the critical sections of the program (the same kind of hatch means that both critical sections refer to the same mutex).

The small numbers near the tasks show the order with which they are executed, alternating between jobs and remaining stuck at the start of critical sections when the related mutex is busy.

- when called, it can accept parameters that change its behaviour;
- once finished, it returns a value that can be used for subsequent computation[21];
- multiple instances of the same function can be run at the same time without conflicting with each other.

**Cycle** it represents the order of execution of the various tasks. The name comes from the fact that, at any one time, it only contains a small portion of the tasks necessary for the program and its content is updated by a loop inside the core function of the scheduler.

ScheMo manages the execution of tasks in a very simple way: it has a loop that, at each iteration, cycles through all started jobs and adds the current task of each to the Cycle (potentially skipping some jobs if they are locked[22] or defined to be run less often) and then proceeds to execute them. Once all jobs have terminated, the scheduler stops[23].

A new job can be started by informing the scheduler of its first task and asking for it to be executed.

Each task then must tell the scheduler what the next task in the job is before terminating (this process can be automated by `schemop`[24]).

ScheMo also incorporates a system to manage shared memory between jobs using mutexes.

A mutex is a dummy variable associated to a block of memory that keeps track of whether or not that block of memory is being used by some job. A job that wants to access that data must make a request to the related mutex, with two possible outcomes:

1. the mutex changes state from *free* to *busy* and the job continues its execution or
2. the mutex remains in the *busy* state and the job locks until the mutex is freed.

To allow other jobs to use the same resources, a job must free the mutex once it has finished accessing the data.

[21] `void` functions are not implemented as of now
[22] See below
[23] It is therefore important to schedule at least one job before starting the scheduler
[24] See below

See Figure 4.15 for a graphical representation of how ScheMo operates.

*Note: ScheMo's management of shared resources is very basic and it is not able to prevent deadlocks or other similar problems related to mutexes.*

### ScheMo Preprocessor (`schemop`)

Even if the C++ library offers various functions and macros to ease the writing of a multithreaded program, complex applications still result in hard-to-read and cluttered code.

To avoid this, we developed a custom preprocessor (`schemop`) that lets the developer write a cleaner code, focusing on the control flow of each job without worrying about defining the various support variables and functions that make the program run smoothly.

`schemop` is written in Python and offers a list of directives[25] that can be used in the source file in place of ScheMo's C++ macros or more complex statements.

The preprocessor also tracks all used jobs, tasks, functions and related variables, automating their definition and initialization.

Lastly, `schemop` allows the management of shared memory by defining *critical sections*, blocks of code that need to access a particular resource. This way, the developer need not to worry about defining, locking and unlocking mutexes since the preprocessor automatically generates the right code to do that.

However it is worthy of note that only statically defined jobs and tasks can take advantage of the full capabilities of this tool, since it cannot reliably predict runtime behaviour. ScheMo allows dynamically created jobs and tasks, but mixing `@`-directives and normal macros is an operation that necessitates a clear understanding of the library and the preprocessor.

Once the source file is given to `schemop`, the latter converts it to a standard C++ source file, ready to be compiled normally with `g++` or any other C++ compiler.

*For an example of ScheMo's features, see Appendix F.*

### ScheMo Profiler

A deeper understanding of the structure of a ScheMo program and the way it is executed can help in its design and optimization.

Because of this, we created a tool (called ScheMo Profiler) that can analyse the program both by looking at the source code and at run-time[26]. The result is a `.profile` file, a text file that is both human-readable and easy to parse automatically.

ScheMo Profiler isn't made of standalone programs, but it is a set of optional features built-in in both the ScheMo Preprocessor and the ScheMo C++ Library. When used in conjunction with the ScheMo Preprocessor, it:

- counts the number of jobs, functions, tasks and mutexes used;
- assigns a name and ID to each job, function and task for additional analysis;
- builds a flow graph of each job and function.

If the profiler features have been enabled during compilation, when the resulting program is run it records in a `.profile` file (either an already existing one or a new one) data about how the scheduler executed the various parts of the program.
Each time the scheduler passes control to a different job, in fact, it adds to the file:

- the ID of the currently running job;

[25] See Appendix C

[26] The current C++ implementation of the profiler uses the `ctime` and `cstdio` libraries, not available in all platforms. Some useful information can still be gathered by using it on the computer used to develop the software instead of the real device.

- the ID of the task the job is currently on;
- a timestamp[27] of when control is passed *to* the job;
- a timestamp[27] of when control is passed *away from* the job.

Using this information it is possible to:

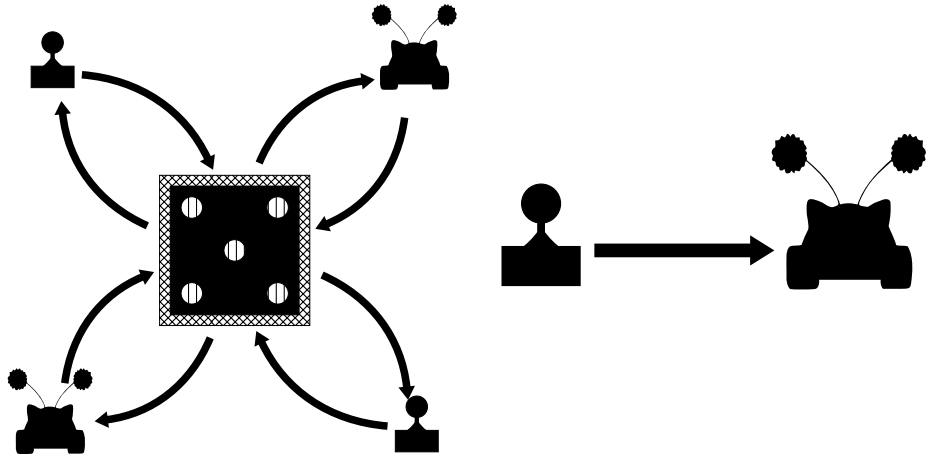
- reveal superflous pieces of code that are never accessed;
- decide whether or not it is convenient to merge or split certain tasks based on their running time;
- identify bottlenecks in the program execution;
- recognise errors in the structure of the program;
- etc...

As of yet, no memory-related profiling has been implemented.

We also created a tool called ScheMoTEX (and in particular the Python script `smp2tikz`[28]) that converts ScheMo Profiler's `.profile` files in LATEX `.tex` files, making it possible to visualize the whole program and more intuitively analyse its structure. Figure 4.15 and the diagrams in Appendix F are examples of ScheMoTEX's output.

### 4.6.2 Network topology

FlipperBot is made of different devices that need to communicate among themselves and for this reason they have to be connected in a suitable network. In particular, the devices can be connected in different ways depending on the current mode: standard game or standalone.



**(a)** Standard Game Network **(b)** Standalone Network

**Figure 4.16:** Network topologies
Board Controller Robot
Arrows go from the client to the server in a connection.

[27] Milliseconds elapsed since the program was launched.
[28] PGF/Ti*k*Z is a LATEX package used in the output files to visualize the graphs

**Standard Game Network (SSID starting with `FlipperBot-Board-`)**
In game mode, the board acts as access point for all other devices, and all communications go through it.

In particular, in this mode controller and robot are not directly connected, instead there is a client-server connection between controller and board and another one between board and robot. This way, the board is able to block or modify the commands given by the controller before sending them to the robot, as suited to make the game work as expected.

The first connection is also used to manage the "everything button" on the controller.

Another client-server connection exists between the robot and the board to allow the transmission of impact information.

This topology is represented in Figure 4.16a.

**Standalone Network (SSID starting with `FlipperBot-Robot-`)**
When no boards are available for connection, a robot can become an access point to create a new network.

A single controller can join this network and create a client-server connection with the robot, directly controlling it.

This topology is represented in Figure 4.16b.

**Isolated Network**
In the case in which neither boards nor robots are available, the controller simply doesn't connect to any other device while waiting for a suitable network to appear.

The board always uses the game mode, broadcasting the network SSID[29] to make its presence known. Robots try to connect to a game network if they can find it, otherwise they get into standalone mode. Controllers connect to whichever suitable network they can find, giving priority to the ones created by a board.

In some cases, a WiFi network can exist and be accessible while it is already *full* (all devices of the previously defined topologies are already present). Such cases are managed by FBCP[30] and lead to changes in topologies.

### 4.6.3 FlipperBot Communication Protocol (FBCP)

In order to create the above networks and share data through them, all devices must use a common language to communicate. For this reason, we created the FlipperBot Communication Protocol (FBCP for short).
It is a plain-text, application-level protocol in which each packet is made of:

- a command identifier;
- zero or more[31] mandatory parameters separated by spaces;
- trailing data that can contain optional parameters but is often ignored;
- a newline character (`0x0A`) to end the packet.

The various FBCP commands are divided in the following groups:
**generic:** common commands that can be used with other groups of commands;

[29] Service Set IDentifier: the *name* of the WiFi network
[30] See 4.6.3
[31] The exact number is dependent on the command

**network building:** manages requests of permission to join a network and grant/denial of it, in addition to changes of network topology;

**robot and controller:** commands specific for communication with robots and controllers;

**debug:** utilities to thelp in the development of new FBCP-compatible software.

To make the use of this protocol as easy as possible, we developed both a C++ and a Python implementation of it.

These tools, however, are only concerned with the creation and parsing of correct FBCP packets, completely ignoring how the device should react to them. In fact, a program that uses FBCP needs not to implement all supported commands but only the ones it is supposed to use, giving an error for everything else.

The C++ library and the Python module are built to be easily extendible, making it simple to add new commands to the protocol.

*A full list of currently supported commands with a short description of each can be found in Appendix D.*

### Controller modes and options

Given the wide range of possible controllers that could be used with this game (the ones described in this text being only a part of them), it can be useful to also have different ways of managing the sharing of information between controller and server (whether it is a robot or a board).

For this purpose, it is possible to set or unset certain *options* for the connection, flags that indicate if certain modifications must be applied to the normal way of communicating. As an example, for the accelerometer-based controller described in 4.2.2, an useful option could be one that removes the need for the controller to send a command to make the robot go forwards, having the server assume that that is the desired direction when no command is sent.

Other than being set or unset, options can, if needed, take a particular value, making them more generic and versatile. There could be, for example, an option called `timeout` that contains information about how long the server should wait for a command before considering the controller disconnected.

In the case that there are many options to be set or unset for the correct use of a controller, it is easier to use *modes*. A mode is nothing more than a predefined collection of options, with their set/unset states (and their additional values if needed) already decided.

Enabling a mode is in no way different from setting the individual options one by one from a functionality point of view, but:

- it is quicker;
- it requires fewer package exchanges;
- it makes it possible to modify the options needed by the server to make a particular controller work without editing the code of the controller (if the name of the mode doesn't change).

A server isn't required to support all possible modes and options, so a controller should:

1. try to select all suitable modes, from the one that allows the best performance to the worst;
2. try to set the necessary options one by one, if the previous point failed;
3. fall back to the standard way of communicating (with direct robot or motor commands) if the available options aren't sufficient.

### 4.6.4 Joystick

The ESP-12 on the joystick is programmed in C++, using the ScheMo library to allow mutiple threads to run simultaneously. The joystick must perform three basic activities at the same time, assosciated to the same number of jobs:

`job_client`: gathers data from the hardware (the button and the two potentiometers) and if:

1. the joystick is connected to another device and
2. the information has changed from the last transmission

it sends the suitable FBCP commands to the server;

`job_link`: manages the blue LED, turning it on and off at the appropriate frequency depending on the current state;

`job_network`: searches for available board or robot networks, tries to connect to them, and puts the joystick in the right state depending on the result, repeating this operation each time it finds itself disconnected.

Since the preferred way to use a controller is through a game network (i.e. connected to a board), the scanned SSIDs are sorted in alphabetical order before trying to connect to them one at a time. This way, connection to the `FlipperBot-Board-...` networks will always be attempted before the `FlipperBot-Robot-...` ones.

While there are specific commands to end a connection, it is possible that a devices breaks the connection without explicitly stating so. This is the case, for example, when a device is turned off (without a source of power it is impossible for it to send the right message).

To prevent problems in such cases, every once in a while the joystick will send a *keepalive* command[32] to the server to check if the device is still actually connected. If no response is received, the joysticks falls back to the idle state and tries to connect to a new network.

### 4.6.5 Robot

The robot's software is similar to that of the joystick in many ways, so only the main differences are listed here.
The jobs executed by the robot are:

`job_client`: it is identical to that of the joystick except that it takes data from the bumper (and thus the sent command is different);

`job_link_and_bumper`: it manages two sets of LEDs instead of only one like the joystick and, in addition to this, it uses information from the bumper to change the current direction[33];

`job_network`: similar to the one in joystick, but only tries to connect to boards, getting into standalone mode if it fails;

`job_server`: each time the robot changes state, it accepts exactly one connection from a device (the board to which the robot is already connected as a client when in game mode, or a new controller when in standalone mode) and executes the received commands related to its direction or the motors speed.

[32] Called `Q_HEARTBEAT` in Appendix D.

[33] These two unrelated functions have been united in the same job only because they are so simple and short that splitting them in two jobs would only be a waste of resources.

Just like the joystick, the robot sends a keepalive signal when necessary to check whether the connection is still active. It is to note, however, that the client job is active only while in game mode (in fact, the joystick wouldn't handle the bumper information even if it received it since it doesn't run a server).

However, unlike the joystick the robot also has a server that needs to assure the functionality of the connection. This is accomplished by setting a timeout (longer than the normal time between a keepalive command and the next) after which the connection is broken if no command is received. Of course, it also replies to incoming keepalive commands.

As said in the list above, the bumper is used to change the direction of the robot (in addition to sending its raw value to the board). In fact, to avoid having the robot needlessly pushing against the walls, the forward direction is prevented when the bumper is pressed (the robot stops instead). Additionally, the forward left and forward right direction are converted to in-place left and right rotations respectively.

### 4.6.6 Board

The Raspberry Pi on the board runs the Raspian Wheezy operating system, a GNU/Linux system built specifically for this device.

All the software we developed to run on the RPi was written in Python, specifically it was tested on version `3.5.2`.

Considering the large number of different, and sometimes unrelated, tasks the board has to carry out, we opted for a very modularized structure. The `flipperbot.board` package, containing all the Python code specifically developed for the board, is in fact divided in the following modules:

- `audio`
- `display`
- `everythingButton`
- `game`
- `led`
- `server`
- `shared`
- `simonsays`
- `totems`

Each of these has a specific function and they have been designed in such a way that their implementation can be radically changed, while their interface to the other modules remains the same. As an example of this, when we switched from an internal to an external display controller, the `display` module has been almost completely rewritten, but not a single line of the other modules needed edits. This means that future improvements of the game will be possible without losing any of the previous work.

Among these modules, the most important is `game`. Its role is to use all other modules to accomplish the common goal of making the game work as wanted. While all other modules[34] concern themselves with managing the interactions with the external world, `game` takes the data retrieved from the former and gives them meaning by implementing the rules of the game, allowing the inputs and outputs to cooperate in a coordinated way.

In addition to these modules, the `board` package contains two subpackages not directly related to making the game work: `debug` and `demo`.

[34] Except for `simonsays`, which contains alternative game rules

### `debug` package

This package was created to facilitate the detection of errors in the code during development and to test the functionality of completed parts.
It's main features are:

- managing the output to screen of informative messages and errors, with the automatic addition of useful information such as the module and class that generated the message;
- writing logs with timestamps and tags that can be used to filter the desired data from the rest;
- visualising a summary of all important data of the game.

The last point is accomplished through the `debug.monitor.Monitor` class, that generates a text-based interface that can be either printed to screen only when desired, or configured to continously display the state of the game and automatically update the information after regular intervals.

This interface can be seen in Figure 4.17.


**Figure 4.17:** `Monitor` interface

Although this interface could in theory be used to play the game (at least in Simon Says mode) without a real board at disposal, a better option would be to use the `demo` package (described below).

### `demo` package

This package was originally created to have a way to test completed parts of the project that needed to interact with hardware that hadn't been built yet. It is made up of different modules:

- `fakeboard`
- `fakedisplay`
- `fakeeb`
- `fakejoystick`
- `fakeled`

Each one of these can work independently from the others, but they all need a real game to be running, either on the actual board or on a different device (e.g. on a PC).

Each module mimics a particular hardware component of the project, offering an intuitive graphical representation of it, as shown in Figure 4.18. If the real counterpart is present (either inside the board or remotely connected to it), the virtual component will show its current status, otherwise it will simulate it.

This interface, however, doesn't only offer visual feedback of what is going on in the game, but also allows interaction with the virtual devices. In fact, it is possible to:

- control the joystick with the keyboard (similarly to the virtual joystick interface shown in 4.2.4);
- press or release the "everything button";
- simulate a hit on a totem.

Although it was created for debugging and testing purposes, this interface (fully or in part) can be used to play the game without needing to mount the real board (saving time and space, but probably losing in fun). Since a simulated robot isn't available, however, only the Simon Says mode can be played this way.

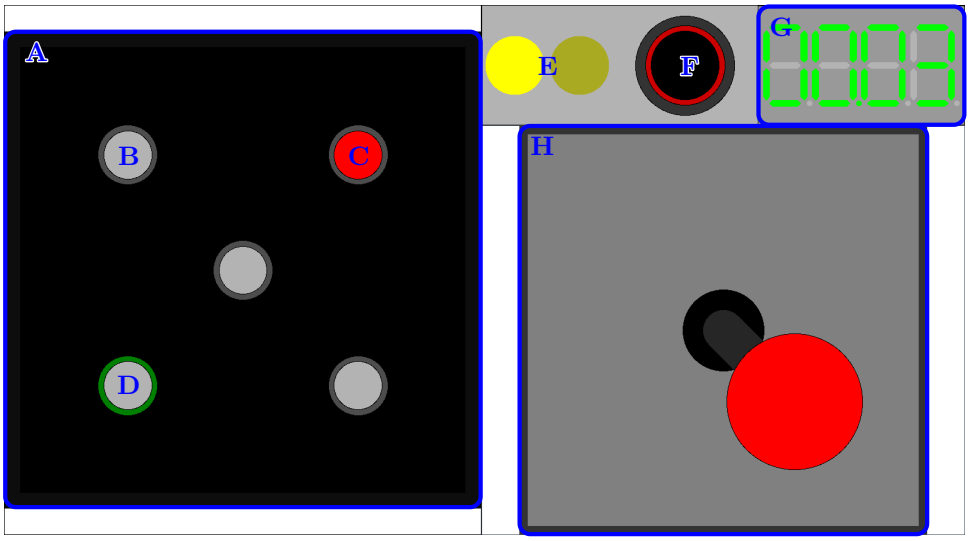



**Figure 4.18:** `demo` interface
**A:** Board; **B:** Turned off totem; **C:** Turned on totem; **D:** Totem that is being hit; **E:** LEDs; **F:** Everything button; **G:** Display; **H:** Joystick.

### External display

As stated in 4.4.1, the display is not directly controlled by the Raspberry Pi on the board but through a dedicated Arduino.

The program loaded on the Arduino is written in C++ and receives commands[35] from an external source (in this case from the Raspberry Pi) through a TTL serial connection with baudrate of 9600 bps.
By sending the right string to the Arduino, it is possible to:

- set the displayed text (if it is longer than 4 character, it will be shown by making it scroll through the display);
- set the refresh rate (i.e. the frequency in Hz at which the characters are "redrawn")[36];
- set the scroll speed (in characters per second);
- enable/disable blinking;
- set the blinking speed.

The program is able to translate various characters into the right set of segments in the display, but if a specific character isn't available it can still be displayed as a combination of the default ones (in the worst case by setting the individual segments).

### 4.6.7 Virtual Interface

As explained in 4.2.4, we developed a virtual interface that can be used instead of the real joystick.

This application is written in Python and uses the Tkinter package for the graphical interface; it can thus be run on a wide range of devices, from PCs to Android smartphones (although not natively for the latter).

Note that this program is not to be intended as a definitive and unimprovable alternative version to the standard controller, but more as a proof of concept. In fact, it shows that the transission of commands is handled in a way that allows for different kinds of controllers, making it possible to use or build the most suitable for the individual user.

This virtual interface takes full advantage of the `flipperbot.fbcp` module to implement the protocol in a simple and efficient way, and the same could be done when developing in C++ instead of Python. While no other languages are currently supported[37], the simple structure of FBCP makes it easy to implement it in a language of choice without any deep technical knowledge.

### 4.6.8 Motor speed analyser

In order to test whether the motors of the robot could move in a synchronised way, using the protocol described in section 5.2, we developed a Python script (using the OpenCV[38] library) that could analyse a video of the wheels spinning and measure the rotation periods.

The analysis is done on a series of frames (extracted from a video by using FFmpeg[39]) with the assumptions that:

1. the wheel axis is almost aligned with the camera;

[35] More details about the structure of these commands can be found in Appendix E.

[36] While higher refresh rates are usually preferred to obtain the illusion of a static text, lower ones can be sometimes used for interesting effects.

[37] While the lack of a Java library seems to indicate the current impossibility to write an Android app, it is theoretically possible to use the C++ library in conjunction with the Android NDK.

[38] `http://opencv.org/`

[39] `https://ffmpeg.org/`

2. two markers of the same color contrasting with the surrounding are placed on the wheel, one on the axis and one on the circumference.

The script tracks these two markers to esteem the orientation of the wheel in each frame, using this information to measure the rotation speed. From now on, the two markers will be called *axis* and *marker* to distinguish them.

There are various configuration options that must be set before starting the analysis to get an accurate result (most of them are represented in Figure 4.19):

**Cropping area:** only this area of the image is used for the analysis, everything else is ignored;

**Starting axis coordinates:** it indicates where in the image the axis is approximately located (during the analysis the axis position is continually updated using the last match to account for slight movements of the camera or the wheel);

**Maximum axis movement:** this is the maximum distance that a detected point can be from the old axis position to be considered a good candidate for the new axis position;

**Minimum marker distance:** this is the minimum distance that a detected point has to be from the old axis position to be considered a good candidate for the new marker position;

**Approximate marker distance:** it esteems the distance of the marker from the axis (length that should remain roughly constant), creating a circumference where the marker is expected to be;

**Marker predominant channel:** it defines whether the color of the marker should be considered blue, green or red during the analysis;

**Marker threshold:** this is the value used to decide whether a pixel of the image is probably part of a marker or not (in the following description of the various steps this will become clearer);

**Approximate marker size:** it indicates how many grouped pixels must be detected *at least* to consider the group a probable marker;

**Pass angle:** this is the angle from the horizontal used to count the rotations and defines the *pass line*.

During the analysis, the following steps are carried out for each frame to measure the current orientation of the wheel (they are shown in Figure 4.20):

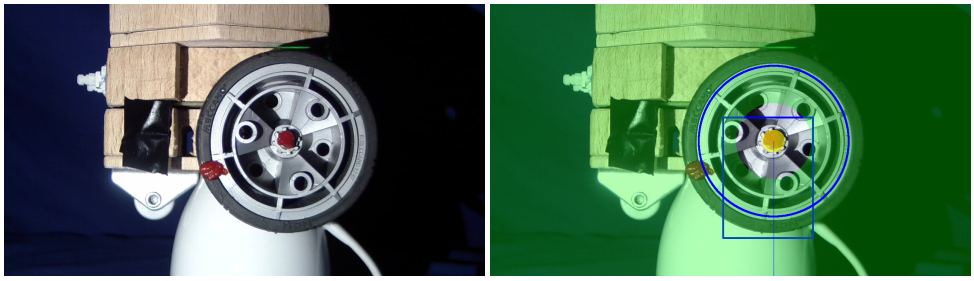
a) the frame is cropped to the cropping area and framed with a black border (to avoid problems with OpenCV's blob detection);

b) a greyscale value is calculated for each pixel, using the formula

$$value = 255 \cdot \frac{c}{r + g + b}$$

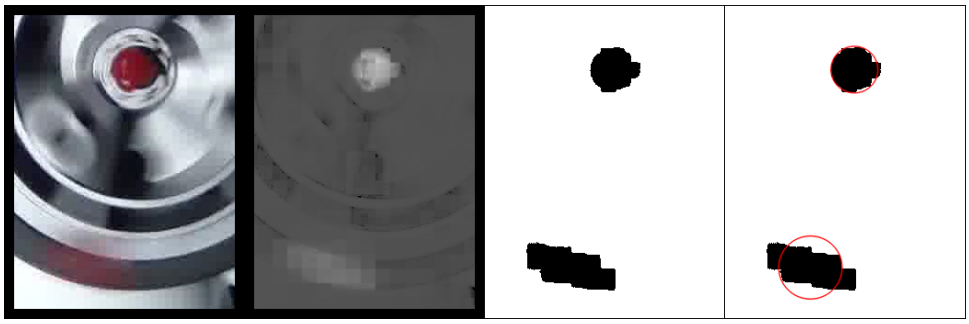
where: $value$ is the greyscale value
$c$ is the value of the pixel corresponding to the marker channel
$r$ is the red channel value of the pixel
$g$ is the green channel value of the pixel
$b$ is the blue channel value of the pixel

c) a threshold is applied to the filtered frame to extract blocks of pixels which color is mostly that of the marker channel;



**(a)** Original image **(b)** Image with configuration options overlayed

**Figure 4.19:** Graphical representation of the configuration options
The blue rectangle indicates the cropping area, the blue circle defines the expected path of the marker, the thin blue line corresponds to the emphpass line, the yellow and green areas are respectively where the axis and the marker can be located.



**(a)** Crop **(b)** Filter **(c)** Threshold **(d)** Blob detection

**Figure 4.20:** Steps of the analysis of a frame

d) the OpenCV `SimpleBlobDetector` is used to detect and locate blocks of continous black pixels (considering the minimum size defined by the configuration options), from which the best matches are extracted.

The best matches are decided according to the rules defined by the configuration options, that is:

- the detected axis position can't be too far from the old axis position;
- the detected marker position can't be too near to the detected axis position;
- if more than one axis candidate is detected, the one nearest to the old axis position is chosen (its position becomes the axis position for the next frame);
- if more than one marker candidate is detected, the one which distance from the axis is closer to the *approximate marker distance* is used.

If both the axis and the marker were succesfully detected, the angle of rotation is estimated from their position:

$$\theta = \operatorname{atan2}\left(M_y - A_y, M_x - A_x\right) - \theta_0$$

where: $\theta$ is the angle of rotation
$A_x$ is the $x$ position of the axis
$A_y$ is the $y$ position of the axis

$M_x$ is the $x$ position of the marker
$M_y$ is the $y$ position of the marker
$\theta_0$ is the pass angle

When, between two frames, the angle changes sign, a new rotation is counted. The time of crossing is then calculated by assuming a constant angular speed between the two frames:

$$time = \frac{1}{FPS}\left(\#frame - \frac{\theta}{\theta - \theta_p}\right)$$

where: $time$ is the time of crossing
$FPS$ is the framerate of the original video
$\#frame$ is the index of the current frame
$\theta$ is the angle of rotation in the current frame
$\theta_0$ is the angle of rotation in the previous frame

By subtracting the time of consecutive crossings, the rotation periods are computed.

*Important note: since this algorithm hasn't been properly validated yet, it is not suggested to trust its results without questions. It should only be used to speed up measures and calculations while supervising it to check if the correct passing frames are detected.*

# 5. Method Evaluation

Some of these tests either would have required too long time or could not be performed due to difficulties in organisation (e.g. too small a sample to have statistically relevant results). We propose here some ideas to verify our assumptions, but unfortunately it was not possible to carry out them all.

Since we chose to focus on realisation, we tried to finish the project by the time given, neglecting a proper planning of measures of success and tests.

## 5.1 Proposed tests

### 5.1.1 Tests During Realisation

#### Battery Regulation Circuit Accuracy

The regulatory circuit should disconnect robot batteries only when power supply drops under 3.3V.

Accuracy could be tested by monitoring at various inputs the outputs of the two triggers (separately). The inputs would be provided by connecting the batteries (individually) to a trimmer and regulating the voltage supply of the circuit. It should be calculated standard deviation and RMS of error (as the difference between the tension at which triggers are activated and 3.3V).

#### Impact Detection

Switches on the robot as well as on the totems should be effectively pushed and signal should reach the board at every impact. The board itself should recognise as effective only simultaneous impacts on both robot and selected totem (with a minimum lag).

Impacts would be simulated on separate parts: robot (1), totem(2) and between the two of them (3). These would be given randomly (e.g. totem, totem, totem/robot, totem, robot, totem/robot etc.) and we record the counting of the programs. We would define accuracy as the percentage or recognised impacts over the total. Robot and totem should recognise respectively (1) and (2) type of impact, while the board should recognise and update score only with type (3).

$$Accuracy = \frac{\#Recognised\ impacts}{\#Total\ impacts\ given}$$

#### Wi-Fi Network Robustness

The network among robot, board and totem should be robust to interferences from other sources (such as nearby computers, cellphones etc.). To test this requirement we would count the logs during the game on the various components and check at the end of the match the number of misunderstood packets.

The protocol created should minimize this problem, for every time a packet is sent there has to be an answer from the receiver, and if there is not, the packet is sent again. Since the network is simple and does not need to protect possibly private information, there was not need to provide it with protection from possible hackings.

### 5.1.2 Tests After Realisation

**Measurement of time played (in minutes)**

For the same person, trend in playing time over sessions should be calculated (if the game is not boring it should not be negative);

For the two populations (single and double player) the playing should be tested to be significantly different (our thesis is that double player population should have a longer playing-time).

**Measurement of the percentage of active interaction of the child in the conversation (not necessarily speaking)**

$$PAIT(Percentage\ of\ Active\ Interaction\ Time) = \frac{T(active\ interaction)}{Total\ time}$$

For active interaction, we mean every attempt of the child to express his/her opinions, or to describe some situations. Since some of the patients may have difficulties in speaking properly, also other forms of communication should be included. For example, if the child uses a vocal synthesizer or a device with images to communicate, the time allotted to use them accounts as active interaction time. Or, more simply, if the child mimics a phase of the game while not explicitly talking, that accounts as active interaction time too.

For the same session, difference in mean PAIT between the registrations taken before and after every session should be checked (our thesis being that if the child is happy with the game, the active interaction time should be longer after sessions).

For the same person, we should also calculate trend, as with playing tome. Our thesis is that is the game is boring, trend should be negative, for the child may loose enthusiasm.

The same considerations for the two populations as above can be applied here to verify if double player mode is more effective in promoting social skills. In this case the coefficients in the trend of PAIT after game session should be confronted.

**Questionnaire**

Questionnaire should be possibly with some numerical responses (such as "How would you rate the child's enthusiasm for playing with others? From 0 (no enthusiasm at all) to 5 (very eager to)"). However, the rating here are personal ad rather qualitative, and probably should be used only to support statistic rescults.

## 5.2 Performed tests

**Motor Coordination**

As previously said, motors on the robot are actually modified servomotors. This may undermine the required synchronism. Therefore, we decided to test both coordination between wheels and, for the same wheel, coordination between clockwise rotation (from now CW) and counter-clockwise rotation (from now CCW).

**Protocol:** The procedure for the evaluations is here described:

1. the period of rotation (in s) was recorded for each wheel for both CW and CCW, for 1 minute of observation;
2. to test normality of the four populations, an Anderson Darling test was performed;

3. for the same wheel, a T-test (in case the distribution resulted normal), or a Wilcoxon Test (in case the distributions deviated from normal) was performed to verify the means of CCW and CW to be equal;
4. for the separate wheel, the same test as above were performed, in order to verify the coordination between left CW and right CCW (forward) and left CCW and right CW (backwards);
5. box plots were drawn to exemplify conclusions for each of the four tests;
6. the same passages were performed after removing outliers using Chauvenet's criterion.

Data were acquired using a Python program written by us to analyse video clips of rotating wheels[1]. The program used red colour marks placed on the centre and on the rim of the wheel to determine in which frame a rotation was completed. Knowing the frame rate of the camera (50fps), it was possible to estimate period of rotation, and the maximum error of measures (20ms).

[1] See 4.6.8

# 6. Results

Observations were:

- for left wheel: 74 CW (no outliers), 81 CCW (10 outliers);
- for right wheel: 60 CW (1 outlier), 87 for CCW (1 outlier);

Unfortunately it appears that motors are not well coordinated. None of the distributions appeared normal, and the situation did not improve after removing outliers. P-values were all in the range of 0.05%.

Also, it appears that the distributions medians are significantly different, and this applies both to the comparison of the same wheel CW and CCW rotation, and in the comparison of the opposite wheels.

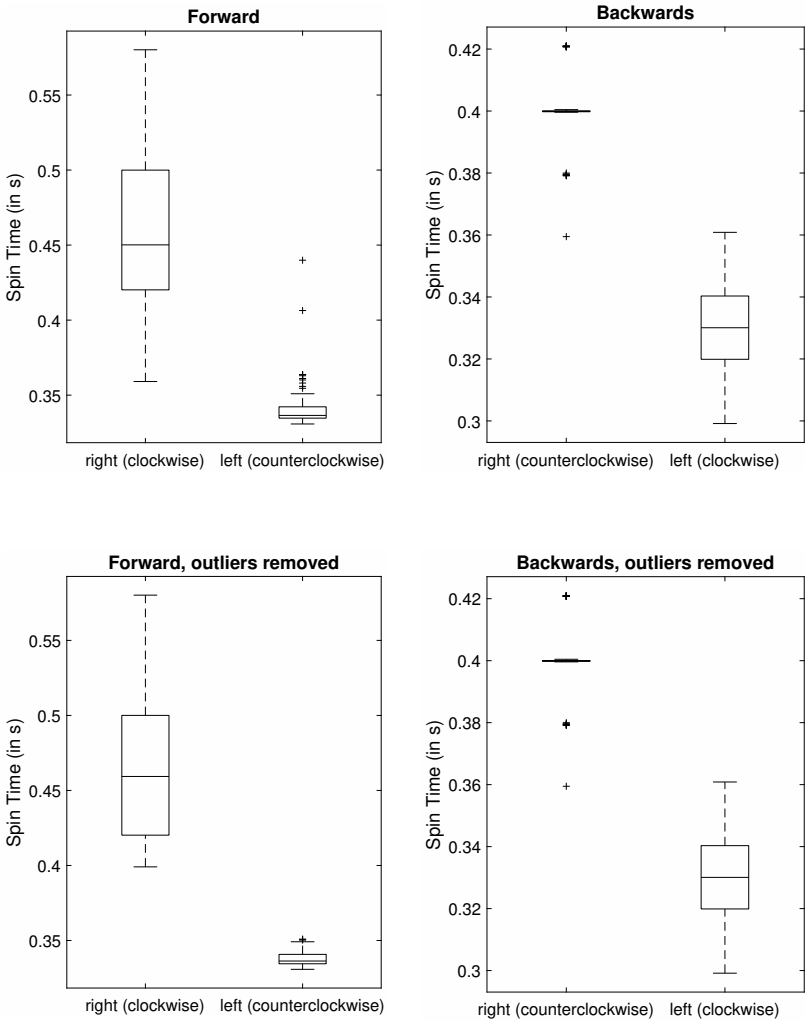



**Figure 6.1:** Boxplots of acquired data

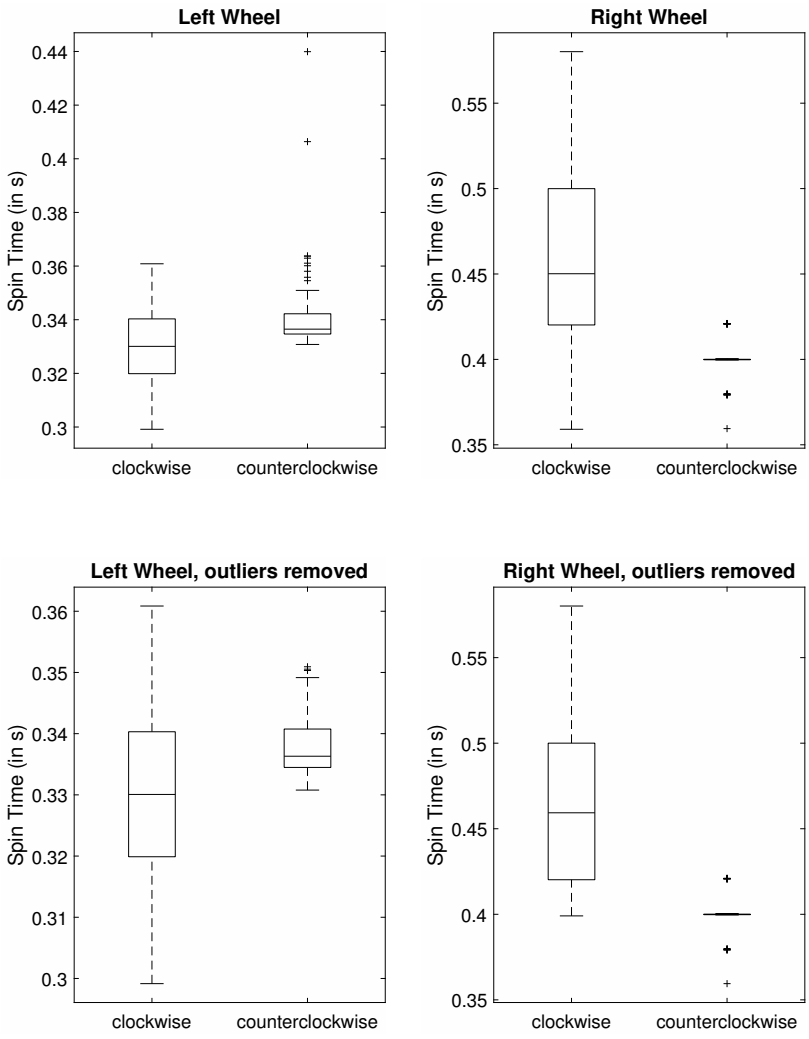



**Figure 6.2:** Boxplots of acquired data (continued)

# 7. Conclusions

It can be inferred that the scarce quality of the result can be ascribed to a consistent slowing in the rotation as time passed by, as shown in Figure 7.1.

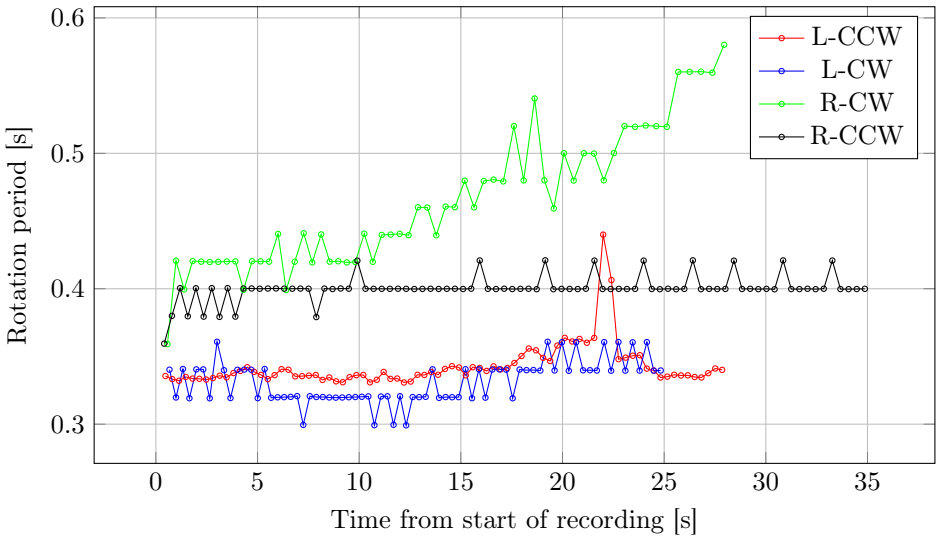



**Figure 7.1:** Plot of the periods of rotation for each sample.

Nevertheless, since for the purpose of the game the robot does not need to move in a straight line for 1 minute, this does not affect the use of it too heavily. In conclusion, we can say that modified servomotors do not make a suitable substitute for regular motors, although it is not so patent during the playing of the game.

# 8. Future Developments

As said in Abstract, sometimes we had to deviate from what we planned in order to have a functioning prototype in time for the deadline. These changes were due to a number of reasons:

- little time available;
- faulty components;
- poor resources available (both because of monetary costs or because of long time needed for components to be shipped);
- programming needed was too complex or too time-consuming.

Nevertheless, it seemed fair to point out where it happened so that it could be discerned what was due to mistakes in design and what difficulties could be avoided instead with more time/resources. Also, this may be useful to anyone who would like to improve our achievements.

## 8.1 Possible Improvements

### The Game

Due to little time, we chose not to develop the second player interface. All the considerations made for the single player mode still apply, the changes should be made mainly to the program.

### The Controller

The controller is the component that resulted most affected by the large variety of possible impairments found in patients. Due mostly to the little time, and second-handedly to the delay in shipping of components, we decided only to fully design and realise the joystick interface, which looked to us as the one who could cover the widest range of limitations.

Extended considerations over the other options are provided in section 4.2.

### The Robot

The second player uses a robot and a controller exactly like ESPer's. These were not realised mainly due to lack of time. Another good idea would be to create multiple "outfits" for the robots, so that players can personalise their own own ESPers.

The current external shell was made to be removable, also to recharge batteries, and could be easily replaced with a different one. This was not done also due to lack of time.

Another interesting development would be to implement a program that allows the robot to choose to disobey to the child sometimes, to give the idea that it is acting on its own will. This should give the child the idea that ESPer is just like a child itself, and needing of his/her support, improving empathy and self-esteem.

### The Board and Totems

We managed to realise the board the way we designed in the time allotted. A possible improvement could be to create a box were to store it properly when disassembled.

We also designed a Totem-Board mating based on magnets to make it more appealing and fun for the child to assemble this part on his/her own. We also tried to

realise one of this matings, but we encountered a number of difficulties and resolved to using jack plugs. These difficulties are due, in our opinion, mostly to the precision needed in the construction and to a lesser extent in finding the right magnets. It could still be made in the right conditions.

## Method Evaluation

We chose to put realisation at the top of priorities, and moreover there was physically no time to perform a proper testing of the game performances. Our intention is to carry out a global testing by letting users play with the game to see if they liked it and how well the various parts behaved.

# 9. Acknowledgements

We would like to set aside this little space to thank those who contributed to our effort.

First of all, our supervisor professor Andrea Bonarini for the precious advice, and Elena De Momi, the teacher responsible for the "Project in Instrumentation and Functional Assestment", as well as the course tutors, Elisabetta Peri and Noelia Chia Bejarano.

A special thank to professor Franco Zappa for helping us out with a faulty component.

Thanks also to Greta, Giada and Giammarco, who contributed to give voice to ESPer.

Finally, we would like to thank all those people that started doubting our own existence during the months dedicated to the FlipperBot experience. Thank you, your patience was priceless.

# Appendices

# A. Circuits and Electric Schemes

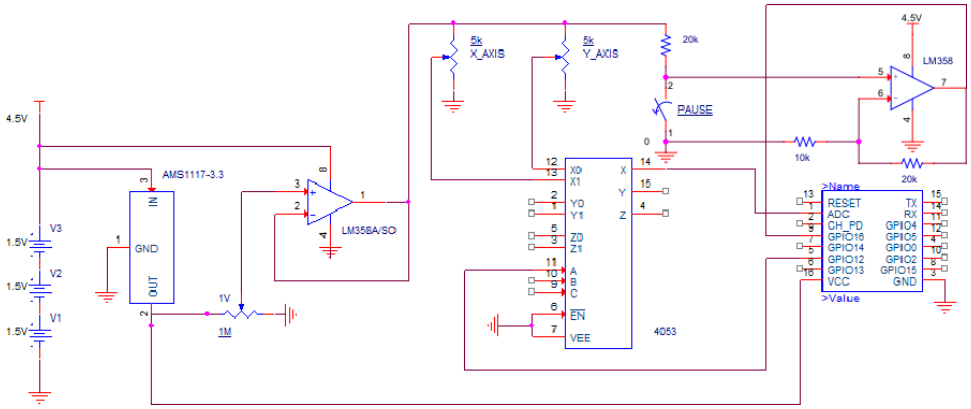



**Figure A.1:** Scheme of Joystick Circuit.

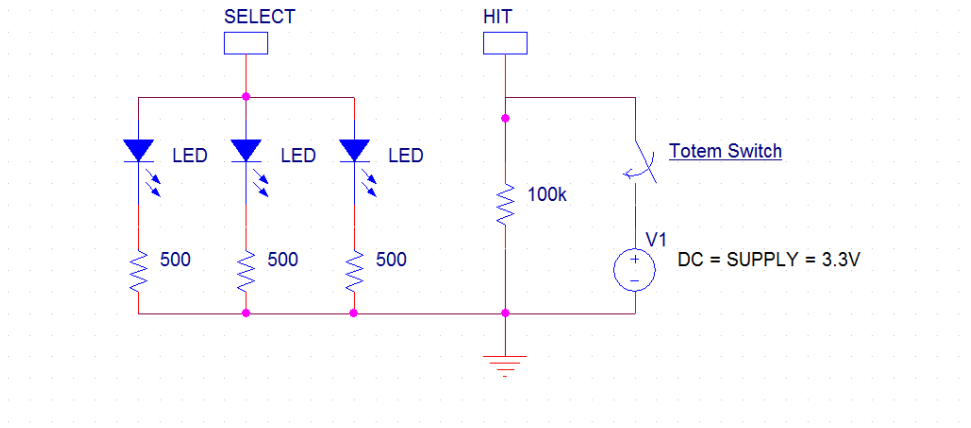


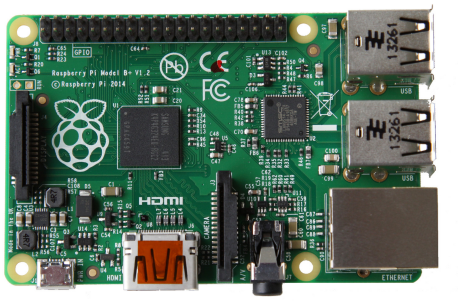

**Figure A.2:** Scheme of Totem Circuit.

# B. External Resources

## B.1 Raspberry Pi (RPi)

Raspberry Pi is responsible for the main part of game-management, being the true centre of the interaction. It deals with:

1. totem lighting;
2. scoring and impact detection;
3. countdown;
4. sound effect;
5. display effect (through Arduino);
6. Wi-Fi communication, receiving from and transmitting to both robot and controller;
7. management of game states.



Raspberry Pi is a single-board computer with a voltage supply of 5V (from the battery pack).

The model used in the game is Raspberry Pi B+.

Features:

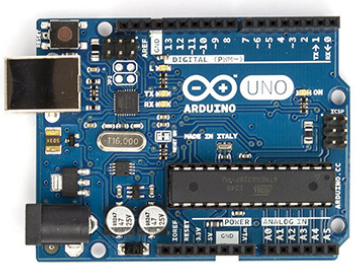
1. 40 GPIOs (General Purpose Input/Output);
2. 4 USB ports;
3. audio output (3.5mm TRRS jack);
4. 15-pin MIPI camera interface (CSI);
5. HDMI, composite video (3.5mm TRRS jack);
6. 10/100 Mbit/s Ethernet (8P8C);
7. 512MB RAM.

GPIOs are used to manage totem lighting and impact detection, as well as LED lights, “everything button” , while 2 USB ports are used for Wi-Fi network and serial communication with Arduino, the audio output is dedicated to sound effects. All programming was done in Python 3. It is located in the main section of the board.

## B.2 Arduino

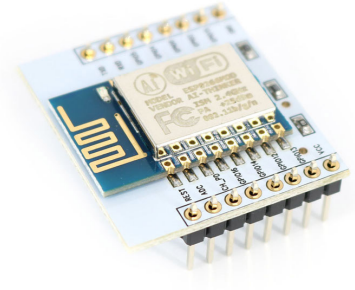
Arduino is a micro controller with its own dedicated IDE and can be programmed using C++. Power supply is 5V. In the project, we used a Arduino UNO as a dedicated controller for managing the display in the main section of the board.
Features:

1. 14 digital input/output pins (of which 6 can be used as PWM outputs);
2. 6 analog inputs;
3. 16 MHz internal clock;
4. 1 USB port;
5. 1 power jack;
6. ICSP header;
7. reset button.




It receives serial inputs from one of the USB ports on the RPi, and uses digital outputs to control display. Inputs correspond to string of characters that should be printed on the display. The deconding of the string is left to the Arduino. It is also placed on the main section of the board.

## B.3 ESP



ESP-12 is an ultra-low power Wi-Fi module, and can be programmed in the same way as Arduino. Its voltage supply is 3.3V, and offers:

1. 16 digital input/output pins (that can be used as PWM outputs);

2. Wi-Fi module;
3. ADC.

We chose a separate controller to use on both robot and controller, for it offered:

- low power consumption;
- smaller dimensions and reduced weight;
- little cost (approximately 2€ apiece);
- embedded Wi-Fi (without having to use an external adapter as in RPi).

**ESP on the controller**

The unit on the controller is responsible for:

- search for connection and transmitting information to the board;
- interpretation of signals from the joystick;
- management of LED lights.

**ESP on the robot**

The unit on the robot is responsible for:

- search for connection and transmitting/receiving information to and from the board;
- driving wheels;
- management of impact detection;
- management of LED lights.

Because of the need to have multiple processes concurrently running, we created a scheduler to simulate parallelism on a single processor[1].

## B.4 Themes and Sounds Effects

All music themes come from royalty free sources:

1. from Creative Common Music Archives:
   - "Nachtwandel" , by Andy G. Cohen — *pause*;
   - "Rainbow Street" , by Scott Holmes — *active play*;
   - "Space Outro" , by Andy G. Cohen — *game over*;
2. "Acquarium" from "Carnaval des Animaux", by Camille Saint-Saëns — *menu*.

Sounds effects (such as beeps and robot squeaks) were created by us by elaborating voice recordings using Audacity. The format used for audio files is Ogg Vorbis, and is reproduced using VLC Media Player.

[1] For details on implementation, see 4.6.1 and Appendix F

# C. `schemop` directives

Table C.1 shows a list of directives currently supported by the ScheMo Preprocessor.

Each directive is identified by a string name (shown in the first column of the table[1]) that is used in conjunction with a prefix to be recognized by `schemop`. There are two default prefixes:

- a *full prefix*: `@SCHEMO_`
- a *shorthand prefix*: `@`

The desired prefix (either one of the default ones or a user-defined one) can be selected using a command-line option of `schemop`.

In the table, the usage strings are written using the shorthand prefix for brevity, but they apply to any other prefix as well.

Inside the usage strings, italicized names are to be replaced with suitable to get the desired result.

**[Table C.1]**

| Directive | Usage and description |
|---|---|
| | *Automatic code generation* |
| `DECLARE` | `@DECLARE`<br>Must be placed in the global space before any other directive.<br>The preprocessor replaces it with the declarations of all the variables and functions needed to make the code generated by the other directives work. |
| `INIT` | `@INIT`<br>Must be used before any scheduling occurs.<br>It initializes all needed variables and the scheduler itself. |
| `SCHEDULE_ALL` | `@SCHEDULE_ALL`<br>Shortcut command to schedule all recognized jobs. Jobs that have been defined without the use of the ScheMo Preprocessor are not automatically scheduled this way.<br>To schedule jobs individually the C++ function `schemo::schedule_job(`*`name`*`)` can be used. |
| | *Job definition* |
| `JOB[1]` | `@JOB (`*`name`*`) {`*`code`*`}`<br>Defines a new job called *`name`*. The *`code`* part contains the definition of the job similarly to a normal C++ function, with the added possibility to use special `schemop` directives. |
| `JOB[2]` | `@JOB {`*`code`*`}`<br>Like the one above, but the job name is autogenerated by the preprocessor. Because of this, the job can only be scheduled by using the `@SCHEDULE_ALL` directive. |

**[Table C.1]**

[1] When present, a number between square brackets is used to distinguish between different uses of the same identifier, but it is not to be considered part of the name

**[Table C.1]**

| Directive | Usage and description |
|---|---|
| JDELAY | `@JDELAY (`*`delay`*`)`<br>If used, it must be placed inside a `JOB` block.<br>Defines how many cycles the job must skip between the execution of one of its task and the next. It can be useful if some tasks need to get control less often than others to work properly.<br>If this directive is not used, a *`delay`* of `0` is assumed (meaning that no cycles are skipped). |
| | *Control flow*[2] |
| TBREAK[1] | `@TBREAK (`*`name`*`)`<br>Explicitly ends a task and starts a new one called *`name`*.<br>It is useful to add break-points where the scheduler can pass control to a different job when long linear pieces of code without interruptions are present. |
| TBREAK[2] | `@TBREAK`<br>Like the one above, but the task name is autogenerated. |
| WHILE[1] | `@WHILE (`*`condition`*`) {`*`code`*`}`<br>`schemop`'s counterpart of a C++ `while` loop. The differences with a normal C++ `while` loop are that:<br>• a break-point is added before the first condition check and before each loop-back, meaning that control is passed to a different job after each iteration<br>• using other control flow directives or a `CALL` directive inside a normal C++ control flow statement would break it |
| WHILE[2] | `@WHILE {`*`code`*`}`<br>Like the one above, but a `true` condition is assumed. It effectively generates an infinite loop. |
| IF | `@IF (`*`condition`*`) {`*`code`*`}`<br>`schemop`'s counterpart of a C++ `if` statement. The reasons to use it instead of a normal C++ `if` are similar to those in favour of `WHILE` statements. |
| ELSE | `@ELSE {`*`code`*`}`<br>`schemop`'s counterpart of a C++ `else` statement to be used after an `IF` block instead of the standard C++ statement. |
| CONTINUE | `@CONTINUE`<br>`schemop`'s counterpart of a C++ `continue` statement to be used inside a `WHILE` block instead of the standard C++ statement. |
| BREAK | `@BREAK`<br>`schemop`'s counterpart of a C++ `break` statement to be used inside a `WHILE` block. |
| EXIT | `@EXIT`<br>Stops the execution of the current job. |

**[Table C.1]**

[2]These directives can only be used inside `JOB` and `FUNCTION` blocks.

**[Table C.1]**

| Directive | Usage and description |
|---|---|
| SHUTDOWN | `@SHUTDOWN`<br>Stops the main scheduler loop and thus the execution of all scheduled jobs. |
| | *Memory management*[2] |
| MEMORY | `@MEMORY {variables}`<br>Defines the variables available to all tasks of a job. Multiple `MEMORY` blocks can be defined for the same job.<br>*variable* is a list of `VAR[1]` directives. |
| VAR[1] | `@VAR (name : type)`<br>Used inside a `MEMORY` block to declare a job-wide variable called *name* of type *type*. |
| VAR[2] | `@VAR (name)`<br>Used outside a `MEMORY` block to refer to a previously declared ob-wide variable. |
| CRITSEC | `@CRITSEC (name) {code}`<br>Protects a block of code with a mutex called *name* (defining the latter if it is its first appearance in the code). |
| | *Functions* |
| FUNCTION | `@FUNCTION (name) options {code}`<br>Defines a function that can be used with the `CALL` directive and can contain control flow directives, similarly to a `JOB` block.<br>*options* is a list of zero or more `PARAM[1]` directives and exactly one `RETURN[1]` directive. |
| PARAM[1] | `@PARAM (name : type)`<br>Defines a parameter for a function when placed among its *options*.<br>*type* and *name* represent respectively the type of the related variable and the name with which it can be referenced inside the function. |
| RETURN[1] | `@RETURN (type)`<br>Defines the type of variable returned by the function when placed among its *options*. ScheMo functions cannot be declared `void`. |
| PARAM[2] | `@PARAM (name)`<br>Used inside a function to refer to the parameter with the same name. |
| RETURN[2] | `@RETURN (value);`<br>Used inside a function to stop its execution and return *value* to the caller. |
| CALL[1] | `@CALL (function) : result;`<br>Calls a ScheMo function with name *function* without parameters and places the returned value in a new variable called *result*.<br>The type of the variable is inferred from the function definition.<br>*result*'s scope is limited to a task that starts right after the `CALL` statement. |

**[Table C.1]**

**[Table C.1]**

| Directive | Usage and description |
|---|---|
| `CALL[2]` | `@CALL (`*`function ; parameters`*`) : result ;`<br>Like the one above, but semicolon-separated parameters are passed to the function. The parameters must be passed in the same order with which they were defined in the function's *`options`*. |

**Table C.1:** List of ScheMo Preprocessor directives

# D. FBCP commands

Table D.1 lists all currently supported FBCP commands.

The first column contains the name with which each command is referred to in code (both using the C++ library and the Python module). The first letter indicates whether the command should be sent from the client (**Q**uestion) or from the server in response to a previous command (**A**nswer).

The second one shows how a packet must be written to be understood as the related command (the words in italic are mandatory parameters and should be replaced with appropriate strings). The terminating *newline* character has been omitted for the sake of readability.

The last column briefly explains how each command is used.

The various commands are divided in the groups described in 4.6.3.

**[Table D.1]**

| Command name | Packet structure | Description |
|---|---|---|
| | *Generic* | |
| `A_ACCEPT` | `ON_IT!` | Sent if the previous command was accepted and correctly handled. |
| `A_REFUSE` | `NOT_NOW` | Sent if the previous command was not handled as requested. |
| `A_ERROR` | `WHAT?` | Sent if the previous command couldn't be understood or wasn't a correct FBCP packet. |
| | *Network building* | |
| `Q_SINGLE_PRESENTATION` | `I'M` *`serial`* | Asks for permission to join a network. *`serial`* is an unique identifier of the device[1]. |
| `Q_MULTI_PRESENTATION` | `WE'RE` *`serial friend`* | Asks for permission of the requesting device and another one to join a network. Used when already connected robots and controllers want to join a game network. The *`friend`* device will have a reserved place in the network but will stillr need to send a `Q_SINGLE_PRESENTATION` to the server. |
| `A_GRANT_ACCESS` | `WELCOME` | Sent if a request to join a network is accepted. |

**[Table D.1]**

[1] It must be in the form `FlipperBot-Board-...` for boards, `FlipperBot-Robot-...` for robots and `FlipperBot-Controller-...` for controllers

**[Table D.1]**

| Command name | Packet structure | Description |
|---|---|---|
| A_DENY_ACCESS | BUSY | Sent if a request to join a network is refused. Usually because there are already enough devices connected. |
| Q_HEARTBEAT | STILL_THERE? | Used to avoid timeouts. |
| A_HEARTBEAT | YEP | |
| Q_CLEAN | GOODBYE | Used to disconnect a device from the network. |
| A_CLEAN | BYE | |
| A_REQUEST_PRESENTATION | WHO? | Sent if the previous command was sent without a prior proper connection. |
| Q_CHANGE_NET | MOVE_TO *net* | Used by a robot to inform a connected controller of the wish to change network. The robot must have already sent a Q_MULTI_PRESENTATION command to the new network to reserve a place in the new network for the controller. The *net* parameter must be set to the SSID of the new network. |
| A_CHANGE_ACCEPT | OK | Sent by a controller to accept a change of network. It must then send a Q_SINGLE_PRESENTATION to the new network to effectively join it. |
| A_CHANGE_DENY | NO | Sent by a controller if it isn't willing to join the new network. In answer to this, the robot must leave the new network (that was joined in combination with the controller) and then either:<br>• leave the old network too and rejoin the new one alone<br>• remain in the old network |

**[Table D.1]**

**[Table D.1]**

| Command name | Packet structure | Description |
|---|---|---|
| | *Robot* | |
| Q_MOTOR_COMMAND[2] | MOTOR *motor direction* | Sets the direction of rotation of a specific motor[3]. |
| Q_ROBOT_COMMAND[2] | BOT *direction* | Sets the direction of motion of the whole robot[3] by changing the speed of both motors (not necessarily in the same way). |
| Q_HIT | OUCH | Sent when the bumper of the robot hits something. |
| | *Controller* | |
| Q_OPTION | OPT *option value* | Sets/unsets (based on *value*[3]) an option. For certain options, the value parameter can contain additional information. |
| A_OPTION_ACCEPT | OK | Sent if the requested option was recognized and correctly set/unset. |
| A_OPTION_DENY | DUNNO | Sent if the requested option wasn't recognized or its state couldn't be changed as wanted. |
| Q_MODE_SELECT | MODE *mode* | Enables a certain mode[3]. |
| A_MODE_ACCEPT | OK | Sent if the requested mode was recognised and enabled. |
| A_MODE_DENY | DUNNO | Sent if the requested mode wasn't recognised or couldn't be enabled. |
| Q_EVERYTHING_ON | PRESSED | Asks to set all available options. |
| Q_EVERYTHING_OFF | RELEASED | Asks to unset all available options. |
| Q_RAW_COMMAND | RAW | Used to send any kind of data with the assumption that the server will correctly handle it and use it to guide the robot. The right mode/options should be selected in advance. |

**[Table D.1]**

[2]These commands are also part of the *Controller* group
[3]See Table D.2 to find available parameters
[3]See 4.6.3 for an explanation of options and modes

**[Table D.1]**

| Command name | Packet structure | Description |
|---|---|---|
| | *Debug* | |
| `A_DATA` | `DATA` | Can be used in various contexts and there isn't a specific way of handling it. It can contain any kind of data as optional parameter with the assumption that the requesting device knows how to interpret it. |
| `A_ALIKE` | `MAYBE` *command* | Suggests a command similar with a similar signature to the received one in case the latter is not recognized. It makes recognizing typos easier. |
| `Q_LIST` | `LIST` *type* | Requests a list of supported commands or options, depending on the *type* parameter[3]. The answer should be a `A_DATA` command with a list of comma-separated commands/options as optional parameter. |
| `Q_HELP` | `EXPLAIN` *command* | Requests a description of how to use the command and/or what it does. The answer should be a `A_DATA` command. The implementation can give a more or less detailed answer that isn't usually suitable for automatic parsing but is more useful for interactive interfaces during debugging. |
| `Q_LOG` | `LOG` | Requests the addition of some data to the server logs. |

**Table D.1:** List of supported FBCP commands

Table D.2 contains a list of all predefined parameter strings to be used in conjunction with the commands in Table D.1.

The first column contains the name with which each parameter is referred to in code (both using the C++ library and the Python module).

The second one shows what the corresponding string is.

The last column explains the meaning of the parameter.

The various entries are divided based on the commands with which they are supposed to be used; the mandatory parameter to which the string refers is indicated

in parenthesis.

**[Table D.2]**

| Parameter name | String | Description |
|---|---|---|
| Q_OPTION (*value*) | | |
| OPTION_SET | ON | Indicate whether the related *option* must be set or unset. |
| OPTION_UNSET | OFF | |
| Q_LIST (*type*) | | |
| LIST_OPT | OPT | Used to list options. |
| LIST_CMD | CMD | Used to list commands. |
| Q_MOTOR_COMMAND (*motor*) | | |
| MOTOR_LEFT | ML | Indicates that the left motor speed must be changed. |
| MOTOR_RIGHT | MR | Indicates that the right motor speed must be changed. |
| MOTOR_BOTH | MLR | Indicates that both motors speed must be changed to the same value. |
| Q_MOTOR_COMMAND (*direction*) and Q_ROBOT_COMMAND (*direction*) | | |
| DIRECTION_FORWARD | FW | Changes the speed of the motor/motors to move the robot forwards. |
| DIRECTION_BACKWARD | BW | Changes the speed of the motor/motors to move the robot backwards. |
| DIRECTION_STOP | STOP | Stops the motor/motors. |
| Q_ROBOT_COMMAND (*direction*) | | |
| DIRECTION_FORWARD_LEFT | FL | Changes the motors speed to make the robot turn left while moving forwards. |
| DIRECTION_FORWARD_RIGHT | FR | Changes the motors speed to make the robot turn right while moving forwards. |
| DIRECTION_BACKWARD_LEFT | BL | Changes the motors speed to make the robot turn left while moving backwards. |
| DIRECTION_BACKWARD_RIGHT | BR | Changes the motors speed to make the robot turn right while moving backwards. |
| DIRECTION_LEFT | SL | Changes the motors speed to make the robot turn left in place. |
| DIRECTION_RIGHT | SR | Changes the motors speed to make the robot turn right in place. |

**Table D.2:** List of standard FBCP parameters

# E. External display messages

The Arduino controlling the display accepts string messages and parses them to understand what text to show on the display and how to do it.

A message is a list of zero or more colon-separated symbols, terminated by a semicolon. A symbol represents what should be shown in one of the four seven-segments units of the display and it can be selected from Table E.2. Each one of the default symbols also have a *dotted* version, enabled by adding a dot after the symbol[1], that lights up the decimal-point dot of the seven-segments unit in addition to the segments that form the symbol. Additionally, symbols can be combined together by placing a + between them[2].

In any point of the message, an *options* command can be inserted. It is delimited by parentheses and contains a list of zero or more semicolon-separated options. In the current version of the program, up to three options can be interpreted (the other ones being ignored if present) and they represent:

1. the refresh rate;
2. the scroll speed;
3. the blink frequency.

An option can either be an empty string (in which case the old value is kept) or a numerical value, optionally with a prefix specifying the numerical base used. If the chosen value is 0, the corresponding feature is disabled. All options are applied as soon as the closing parenthesis is encountered.

*Note: if a base is specified but no number is inserted, the value of the option is treated as 0.*

A formal definition in Extended Backus-Naur form[4] of this structure can be seen in Figure E.1.

```
message        = symbol, { ":", symbol }, ";";
symbol         = simple-symbol, [ "+", symbol ];
simple-symbol  = ( "" | ? A symbol from Table E.2 ? ), [ "." ], [ options ];
options        = "(", option, {";", option }, ")";
option         = "" |
                 base, { digit } |
                 non-base-digit, { digit };
base           = "b" | (* Binary *)
                 "o" | (* Octal *)
                 "d" | (* Decimal *)
                 "x";  (* Hexadecimal *)
digit          = non-base-digit | "b" | "d";
non-base-digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" |
                 "a" |       "c"       | "e" | "f" |
                 "A" | "B" | "C" | "D" | "E" | "F";
```

**Figure E.1:** Formal grammar definition of display messages

Actually, the program tries to parse any message it receives, even if it isn't correctly formatted; thus, it is possible to obtain the desired result using a slightly different syntax than the one presented, but the behaviour is clearly defined only for correctly formatted messages.

[1] E.g. "`0`"→ "`0.`".
[2] So that "`r+J`" has the same meaning of "`d`".

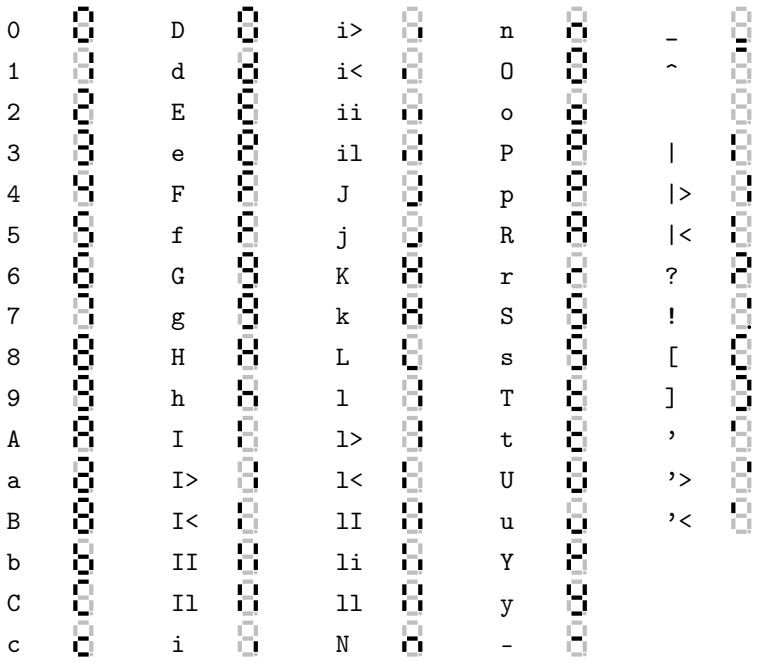



**Table E.2:** List of displayable symbols

# F. Code Snippets

## F.1 C++

In the following pages a reduced version of the program loaded on the robot is presented, used as an example both to explain some of the features of ScheMo and FBCP, and to show how those features can be applied in practice.

This code is not compilable by itself, but shows significant parts of the original program, all extensively commented.

The functions related to the Arduino IDE and the `ESP8266WiFi` library are mostly left uncommented, since an explanation of those tools is out of the purpose of this document.

```
1  #include "schemo.h"      //ScheMo C++ Library
2  //#include "fbcp.h"      //FBCP C++ Library
3  #include "fbcp.common.h" // Additionally includes default
4                           // handlers for common commands
5  #include "FBNet.h"       //IP addresses and ports used
6  #include <ESP8266WiFi.h> //From the ESP8266 SDK
```

```
40 /*
41  * Every FBCP-compliant program should define its own serial code. A
42  * suitable variable is already declared as extern in fbcp.h.
43  */
44 fbcp::string fbcp::serial;
```

```
95  typedef enum
96  {
97   READ_TIMEOUT,
98   READ_SUCCESS,
99   READ_FAIL
100 } READ_RESULT;
101 /* Note that types definitions are put before the @DECLARE directive.
102  * This is necessay because there are ScheMo-related entities that use
103  * them (e.g. the function read_command returns a variable of type
104  * READ_RESULT)
105  */
106
107 /*
108  * This declares all variables and functions needed by ScheMo to make
109  * the program work properly. Since all declarations happen here, the
110  * order in which the jobs and functions are defined is not important
111  * even if they refer to each other
112  */
113 @DECLARE
114 //All other directives should be AFTER it.
115
116 /*
117  * Here is a function definition.
118  * The first line must always defines the name of the function. This
119  * name shouldn't be accessed directly and only with a @CALL directive.
120  * All @PARAM lines are optional and define a parameter of the
121  * function by setting the name with which they can be accessed inside
122  * the function and the type of variable in which they are stored. The
123  * order in which they are written is important since they are passed
124  * to the function in that order.
125  * The @RETURN line is mandatory and must be unique. It doesn't have
126  * to be the last one but it makes the code more readable. It defines
127  * the type of variable returned by the function. All comments are
128  * removed by schemop before parsing the source file and this allows
129  * to insert them anywhere without causing problems. This is
130  * particularly useful for writing documentation comments AFTER the
131  * function signature like in this example.
132  */
```

```
@FUNCTION (read_command)
@PARAM (sock    : WiFiClient*)
@PARAM (cmd     : fbcp::COMMAND_LINE*)
/* The fbcp::COMMAND_LINE structure contains parsed data about a full
 * FBCP command string.
 */
@PARAM (timeout : unsigned long)
@RETURN (READ_RESULT)
/**
 * Tries to read a command from the socket @PARAM(sock) and
 * stores it in *@PARAM(cmd) if successful.
 * The possible return values are:
 * READ_SUCCESS : A valid FBCP command was received and
 *                correctly parsed
 * READ_TIMEOUT : @PARAM(timeout) time without receiving a
 *                complete FBCP command
 * READ_FAIL    : Protocol error, no valid FBCP command was
 *                received
 */
{
 @MEMORY
 {
  /*
   * This is a memory block. All variables with job/function-wide
   * scope must be put here.
   *
   * The following directive declares a variable called msg of type
   * fbcp::string. The fbcp::string class is a reduced version of
   * std::string, only implementing the necessary features and
   * allowing the addition of non-standard ones if future developments
   * of the FBCP implementation require them.
   */
  @VAR (msg : fbcp::string)
 }
 @MEMORY
 {
  /*
   * Multiple memory blocks can be put in the same job/function block
   * and they will be joined together by schemop.
   */
  @VAR (t : unsigned long)
 }
```

```
 /*
  * This @RETURN directive is not the same as the one used in the
  * function definition (notice the semicolon at the end). It
  * specifies what value should be returned to the caller and stops
  * the function. More than one of this directive can be (and usually
  * is) present in the same function.
  */
 @RETURN(            // This is the standard way to pars ean FBCP
  fbcp::parseCommand(// command contained in a fbcp::string. The
   @VAR(msg),        // result is stored in the second parameter which
   *@PARAM(cmd)      // is of type fbcp::COMMAND_LINE&. I also returns
  )                  // whether or not the parsing was successful.
  ?READ_SUCCESS:READ_FAIL
 );
}
```

```
/*
 * This is a job definition. Its name can be used to either schedule
 * or deschedule it.
 */
@JOB (job_network)
/**
 * Job that manages the connection to a network.
 * When the robot is idle (neither in game mode nor in standalone
 * mode), it first tries to connect to a board and pass to game mode,
```

```
249  * if it fails it gets into standalone mode.
250  */
251 {
252  // Memory blocks in jobs are identical to the ones in functions
253  @MEMORY
254  {
255   @VAR(i:int)
256   @VAR(n:int)
257
258   @VAR(t1:unsigned long)
259
260   @VAR(cmd:fbcp::COMMAND_LINE)
261  }
262
263  @WHILE // This is an infinite loop
264  {
265   @IF (mode == MODE_IDLE) // Using a ScheMo @IF here allows to use
266   {                       // other directives inside
270    /*
271     * The (true) argument makes this function asynchronous, making it
272     * possible to write the following @WHILE statement to make the job
273     * wait for scan completion without blocking other jobs. This shows
274     * the real importance of using @WHILE directives instead of normal
275     * while statements.
276     */
277    WiFi.scanNetworks(true);
278    @WHILE (
279     (@VAR(n) = WiFi.scanComplete())
280     == WIFI_SCAN_RUNNING
281    ) {}
282    /*
283     * Notice the use of a job-wide @VAR variable inside the loop,
284     * making it possible to still use it in the following code.
285     */
298    @IF (@VAR(n) > 0)
299    {
302     @VAR(i) = -1;
303     @WHILE (++@VAR(i) < @VAR(n))
304     {
324      /*
325       * BOARD_PREFIX is included in the FBCP library instead of the
326       * FBNet one because it refers to the prefix of the serial code
327       * of the board and thus it is part of the protocol.
328       * The fact that it is also used for the SSID of the
329       * board-related network is just for convenience.
330       */
331      if (ssid.startsWith(fbcp::BOARD_PREFIX))
332      {
333       Serial.println(F("Found suitable board network"));
334      }
335      else
336      {
337       /*
338        * @CONTINUE and @BREAK directives must be placed inside @WHILE
339        * blocks, but they can be contained inside normal C++ blocks
340        * inside of them such as this if-else statement. It's important
341        * to note however that they refer to the innermost @WHILE loop,
342        * not any loop.
343        * @WHILE (true)
344        * {
345        *  while (true)
346        *  {
347        *   if (condition)
348        *   {
349        *    break; //stops the while loop
```

```
350         *   }
351         *   else
352         *   {
353         *    @BREAK //stops the @WHILE loop
354         *   }
355         *  }
356         * }
357         */
358        @CONTINUE
359       }
```

```
424       if (!sockOut.connect(gateway, FBNet::PORT))
425       {
426        Serial.println(F("Can't connect to server"));
427        @CONTINUE
428       }
```

```
432       /*
433        * Here is shown how to obtain a valid FBCP command string. As
434        * defined above, @VAR(cmd) is a fbcp::COMMAND_LINE, a structured
435        * representation of an FBCP command string.
436        * The .command member is of type fbcp::COMMAND* and specifies
437        * the type of command used; all FBCP commands are available in
438        * the library and accessible with their name.
439        * The .params member contains all mandatory parameters, mapped
440        * by their name to the actual parameter string.
441        * Additional parameters added to .params are ignored, but they
442        * can be put in a third member (unused here) called .other which
443        * content is simply appended to the end of the command line.
444        */
445       @VAR(cmd).command = &fbcp::Q_SINGLE_PRESENTATION;
446       @VAR(cmd).params["serial"] = fbcp::serial;
447       /*
448        * The actual string to be sent out is generated using the
449        * fbcp::writeCommand function. In case invalid data are supplied
450        * (e.g. missing mandatory parameters) an empty string is
451        * returned instead.
452        */
453       fbcp::string s = fbcp::writeCommand(@VAR(cmd));
```

```
456       sockOut.flush();
457       sockOut.print(s.c_str());
458
459       /*
460        * This is a ScheMo function call. The function will temporarily
461        * take the place of the job from the scheduler point of view,
462        * being treated as the latter. The function, however, isn't
463        * really part of the job and can't access job-wide variables.
464        */
465       @CALL(
466        read_command; // Function to be called
467        &sockOut;          //
468        &@VAR(cmd);        // Parameters
469        fbcp::HARD_TIMEOUT //
470       ) : understood; // Return value, its type is defined by the
471                       // function definition (in this case READ_RESULT)
472       /*
473        * The return value has scope in the task right after the
474        * function call, making it accessible here. If a ScheMo control
475        * flow directive is used, however, it will go out of scope.
476        */
477       if (understood == READ_FAIL)
478       {
479        Serial.println(
480         F("Couldn't understand server response")
481        );
482       }
483       else if (understood = READ_TIMEOUT)
```

```
484       {
485        Serial.println(F("Server timed out"));
486       }
487       else if (
488        @VAR(cmd).command->code      // This is the correct way to check
489        == fbcp::A_GRANT_ACCESS.code // for command equality
490       )
491       {
492        Serial.println(F("Server allowed connection"));
493        Serial.println(F("Connection estabilished"));
494        mode = MODE_GAME;
495        @BREAK
496       }
497       else if (
498        @VAR(cmd).command->code
499        == fbcp::A_DENY_ACCESS.code
500       )
501       {
502        Serial.println(F("Server refused connection"));
503       }
504       else
505       {
506        Serial.println(
507         F("Server answered something strange")
508        );
509       }
510
511       sockOut.stop();
512       Serial.println(F("Connection failed"));
513      }
514     }
```

```
517     if (mode != MODE_GAME)
518     {
519      /*
520       * If the robot failed to connect to a board and get in game mode,
521       * it passes to standalone mode and makes itself visible to
522       * controllers by becoming a WiFi access point.
523       */
524      Serial.print("Starting AccessPoint: ");
525      ssid = fbcp::serial;
526      Serial.println(ssid.c_str());
527      WiFi.softAP(ssid.c_str());
528      WiFi.softAPConfig(host, gateway, submask);
```

```
531      mode = MODE_STANDALONE;
532     }
533    }
534   }
535  }
```

```
1040  void setup()
1041  {
```

```
1062   /*
1063    * As already stated above, it's important to define a serial code
1064    * before starting any communications.
1065    */
1066   fbcp::serial = fbcp::ROBOT_PREFIX;
1067   fbcp::serial += serial;
1068   mode = MODE_IDLE;
```

```
1077   /*
1078    * This initializes both the scheduler and all variables containing
1079    * job/function/task data.
1080    * While @DECLARE should be the first directive to be written in the
1081    * source file and has only effect at compile time, @INIT should be
1082    * placed in such a way that its generated code is executed before
1083    * any other schemop-generated code.
```

```
1084   */
1085  @INIT
```

```
1090  /*
1091   * Here two ways to schedule a job are shown.
1092   * The first one uses a C++ function and adds a single job to the
1093   * scheduler queue, the second one uses a schemop directive and
1094   * schedules all tracked jobs (i.e. the ones defined with @JOB).
1095   */
1096  //schemo::schedule_job(job_network)
1097  @SCHEDULE_ALL
```

```
1100  /*
1101   * The following function actually starts the scheduler.
1102   * All code written inside functions, jobs or tasks is actually
1103   * executed inside this function.
1104   */
1105  schemo::start_cycle();
1106  /*
1107   * Code put here will be executed after ALL jobs have terminated.
1108   */
1109 }
1110
1111 void loop() {}
```

Moreover, to report an example of ScheMoTEX, Figure F.1 shows the generated diagrams of some components of this same program. In particular, the first function is also (partly) present in the above code.

Other components of the program haven't been inserted because they couldn't fit in pages without becoming illegible.

## F.2 Python

In the passages below there are some excerpts from the `flipperbot.board.server` module used in the board. They show the main features of the `flipperbot.fbcp` module, the Python version of the FBCP C++ library.

Since most of these features have already been explained in the C++ example, the conversion of data structures and functions from C++ to Python is indicated by comments in the form `# Cpp_version -> Python_version` placed above the related piece of code, with only significant differences explicitly stated.

```
4  # fbcp.h -> fbcp
5  from .. import fbcp
6  # fbcp.common.h -> fbcp.common
7  from ..fbcp import common
8  from .shared import ThreadEx # An extended version of threading.Thread
9                               # that adds features such as pausing,
10                              # resuming and stopping the thread.
```

```
18 # This function loads information about FBCP commands and default
19 # parameters from a configuration file, making it easier to add custom
20 # ones without directly editing the module source files.
21 # If no configuration file is supplied, the default one is used.
22 fbcp.loadCommands()
23
24 class ClientThread (ThreadEx):
25 """ Represents a new client connected to the server.
26
27 Manages the FBCP connection and any subsequent communication, sending
28 the relevant data to the game module.
29 """
30
```

```
70  def setup(self):
72   # fbcp::COMMAND_LINE -> fbcp.CommandLine
73   self.cmdIn = fbcp.CommandLine()
74   self.cmdOut = fbcp.CommandLine()
121 def manageConnection(self):
122  buf = self.sockIn.recv(256).decode("UTF-8")
124  # fbcp::parseCommand -> fbcp.CommandLine.parse
125  if not self.cmdIn.parse(buf):
126   self.debug("Can't understand message")
127   # fbcp::COMMAND_LINE.command -> fbcp.CommandLine.command
128   # Note that it is an object instead of a pointer (thus normal
129   # assignment by reference can be used)
130   #
131   # fbcp::COMMAND -> fbcp.Command
132   # While in the C++ library commands are accessible from the fbcp
133   # namespace, in the Python module they are contained in the
134   #fbcp.Command class.
135   self.cmdOut.command = fbcp.Command.A_ERROR
136   # fbcp::writeCommand -> fbcp.CommandLine.write
137   s = self.cmdOut.write()
138   self.sockIn.send(s.encode("UTF-8"))
139   self.debug("Sent:", s)
140   return
141  if (
142   # The == operator (__eq__) has been overloaded to allow for the
143   # correct comparison of two Command objects
144   self.cmdIn.command
145   == fbcp.Command.Q_SINGLE_PRESENTATION
146  ):
147   self.debug("Client asked for access")
148   # fbcp::COMMAND_LINE.params -> fbcp.CommandLine.params
149   self.serial = self.cmdIn.params['serial']
150   accepted = False
151   # fbcp::ROBOT_PREFIX -> fbcp.ROBOT_PREFIX
152   if self.serial.startswith(fbcp.ROBOT_PREFIX):
163     self.debug("Robot connected:", self.serial)
164     accepted = True
208  self.cmdOut.command = (
209   fbcp.Command.A_GRANT_ACCESS if accepted
210   else fbcp.Command.A_DENY_ACCESS
211  )
212  s = self.cmdOut.write()
213  self.sockIn.send(s.encode("UTF-8"))
214  self.debug("Sent:", s)
215  if accepted:
216   self.connected = True
217   return
```

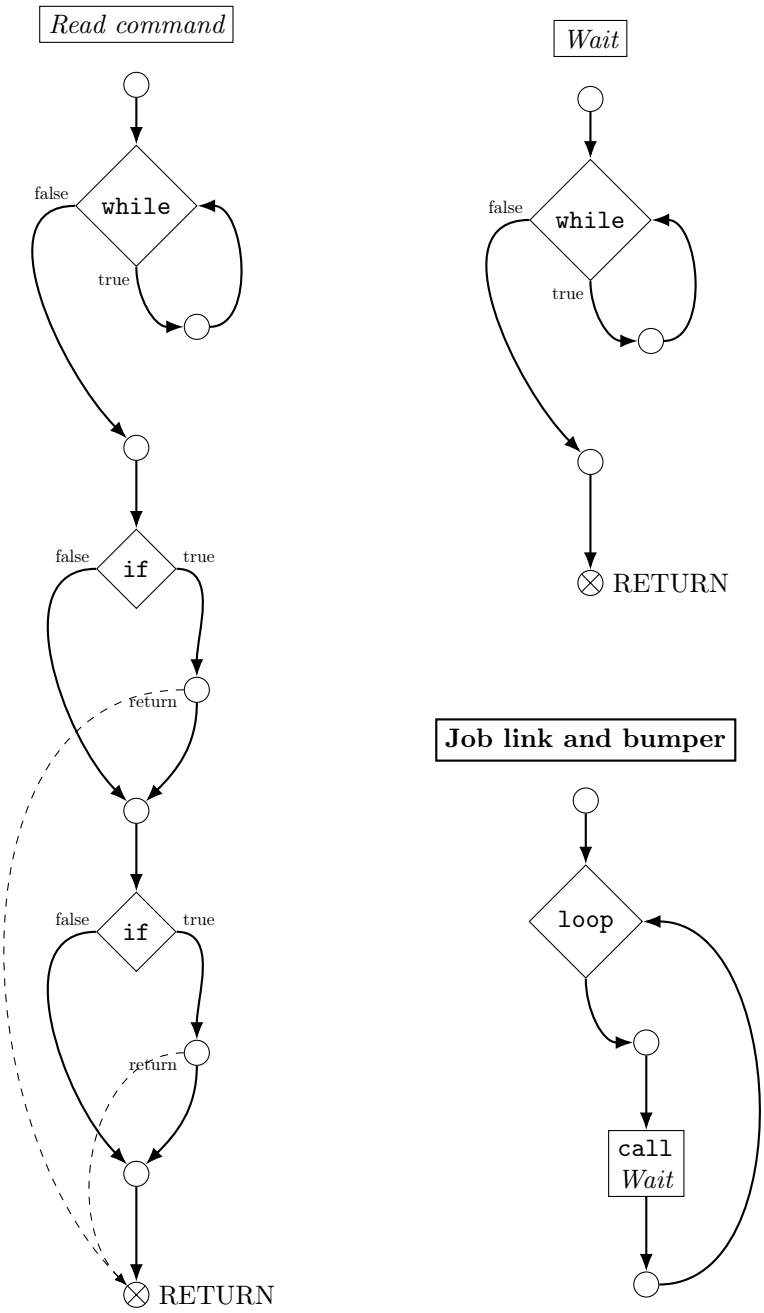



**Figure F.1:** Partial flow diagram of ESPer's program

# Bibliography


[1] John-John Cabibihan, Hifza Javed, Marcelo Ang Jr., and Sharifah Mariam Aljunied. Why robots? a survey on the roles and benefits of social robots in the therapy of children with autism. *International Journal of Social Robotics*, 5(4):593–618, November 2013.

[2] Barbara Celani. Lo sviluppo é gioco... e viceversa. *Tiflologia per l"integrazione*, 3:158–154, 2006.

[3] Kenneth R. Ginsburg and Regina M. Milteer. The importance of play in promoting healthy child development and maintaining strong parent-child bonds. *Pediatrics*, 119(1), January 2007.

[4] International Organization for Standardization. Information technology – Syntactic metalanguage – Extended BNF. ISO/IEC 14977:1996(E), Geneva, Switzerland, December 1996.

[5] Rianne Jansens, Pedro Encarnaçāo, and Serenella Besio. Ludi: a pan-european network addressing technology to support play for children with disabilities. *Proceedings New Friends 2015 - The 1st International Conference on Social Robots in Therapy and Education*, pages 6–7, 2015.

[6] Ivana Kruijff-Korbayovà, Elettra Oleari, Clara Pozzi, Francesca Sacchitelli, Anahita Bagherzadhalimi, Sara Bellini, Bernd Kiefer, Stefania Raciopp, Alexandre Coninx, Paul Baxter, Bert Bierman, Olivier Blanson Henkemans, Mark Neerincx, Rosemarijn Loije, Yiannis Demiris, Raquel Ros Espinoza, Marco Mosconi, Piero Cosi, Rémi Humbert, Lola Ca namero, Hichem Sahli, Joachim de Greeff, James Kennedy, Robin Read, Matthew Lewis, Antoine Hiolle, Giulio Paci, Giacomo Sommavilla, Fabio Tesser, Georgios Athanasopoulos, Georgios Patsis, Werner Verhelst, Alberto Sanna, and Tony Belpaemed. Let's be friends: Perception of a social robotic companion for children with t1dm. *Proceedings New Friends 2015 - The 1st International Conference on Social Robots in Therapy and Education*, pages 32–33, 2015.

[7] Peter Ljunglöf, Britt Claesson, and Ingrid Mattsson Müller. Lekbot: A talking and playing robot for children with disabilities. *Proceedings of the 2nd Workshop on Speech and Language Processing for Assistive Technologies*, pages 110–119, July 2011.

[8] Peter Ljunglöf, Staffan Larsson, Katarin Mühlenbock, and Gunilla Thunberg. Trik: A talking and drawing robot for children with communication disabilities. *Proceedings of the 17th Nordic Conference of Computational Linguistics NODALIDA 2009*, 4:32–33, May 2009.

[9] Laura Marchal-Crespo, Jan Furumasu, and David J. Reinkensmeyer. A robotic wheelchair trainer: design overview and a feasibility study. *Journal of Neuro-Engineering and Rehabilitation*, pages 7–40, 2010.

[10] Cheryl Missiuna and Nancy Pollock. Play deprivation in children with physical disabilities: The role of the occupational therapist in preventing secondary disability. *American Journal of Occupational Therapy*, 45:883–888, 1991.

[11] Yvette Pearson and Jason Borenstein. The interventionof robot caregivers and the cultivation of children's capability to play. *Science and Engineering Ethics*, 19(1):123–127, September 2011.

[12] Renée van den Heuvel, Monique Lexis, and Luc de Witte. Possibilities of the iromec robot for children with severe physical disabilities. *Proceedings New Friends 2015 - The 1st International Conference on Social Robots in Therapy and Education*, pages 46–47, 2015.

[13] Jacqueline Kory Westlund, Leah Dickens, Sooyeon Jeong, Paul Harris, David DeSteno, and Cynthia Breazeal. A comparison of children learning new words from robots, tablets, & people. *Proceedings New Friends 2015 - The 1st International Conference on Social Robots in Therapy and Education*, pages 26–27, 2015.

[14] Frances Wijnen, Vicky Charisi, Daniel Davison, Jan van der Meij, Dennis Reidsma, and Vanessa Evers. Inquiry learning with a social robot: can you explain that to me? *Proceedings New Friends 2015 - The 1st International Conference on Social Robots in Therapy and Education*, pages 24–25, 2015.