

TECHNICAL UNIVERSITY OF VALENCIA



DEPARTAMENT OF COMPUTING ENGINEERING

Master thesis

DroidStorm: Development of a Bluetooth based mobile application for autonomous systems

Tomás Tormo Franco

Advisors:

Dr. Pietro Manzoni

Dr. Juan Carlos Cano Escribá

CONTENT INDEX

1. INTRODUCTION.....	3
1.1 MOTIVATION AND OBJECTIVES.....	4
2. SYSTEM ARCHITECTURE.....	6
3. HARDWARE ENTITIES.....	9
3.1 HTC DREAM.....	9
3.2 LEGO MINDSTORMS NXT.....	11
3.2.1 Programming.....	12
3.3 NXT MODULATED IR EMITTER.....	13
3.3.1 Circuit main components.....	14
The 555 Integrated Circuit.....	14
Resistors and capacitors.....	15
3.3.2 Infrared led.....	16
3.4 HITECHNIC IR SEEKER V2.....	16
4. APPLICATION ARCHITECTURE.....	19
4.1 LOW LEVEL LAYER: COMMUNICATIONS IMPLEMENTATION.....	20
4.1.1 Bluetooth library (BTManager).....	20
4.1.2 The Lego MINDSTORMS NXT Communication Protocol.....	31
SetOutputState command.....	32
4.2 HIGH LEVEL LAYER: USER INTERFACE.....	37
4.2.1 Synchronization mode.....	37
4.2.2 Follower mode.....	45
5. SMARTPHONE AND ROBOT INTERACTION.....	49
5.1 CONTROL BY MOVEMENT.....	51
5.2 CONTROL BY BUTTONS.....	55
6. FOLLOWER IMPLEMENTATION: IR FOLLOWER.....	57
6.1 ROBOT PROGRAMMING.....	57
7. TESTS AND RESULTS.....	62
8. CONCLUSION.....	70
9. REFERENCES.....	72

1 INTRODUCTION

Autonomous systems are widely used nowadays. Its applications covers from factory automation or space exploration, to home robots aimed to address real-life problems in households or workplace environments.

Home robots presence is increasing in daily life, as their level of autonomy and intelligence improves. Some examples are the Roomba [5], an autonomous robotic vacuum cleaner, or the Sony Aibo [6], an autonomous system which emulates a dog and is used mainly for entertaining purposes.

One of the most important drawbacks of this home robots is their interaction with humans. In order to interact with robots, humans usually use artificial communication means such as a computer or joystick, which make the human robot interaction unnatural. In most of the cases, an specifically designed remote control has to be used in order to interact with such robots.

Aiming to facilitate man-machine interaction in a direct and intuitive way, auditive and gesture based interfaces have been a very important research topic in many corporations and laboratories recently. Since there are more than 5000 million phone users all over the world, the use of mobile phones in human-robot interaction, is considered suitable for home robot control purposes. Current phones provide enough computation and communication performance to ensure a proper robot control without the need of additional devices.

Since the mobile phone appeared, a lot of research has been carried out in this area: some approaches proposed the use of a mobile phone to receive voice commands to control a robot [1]. This system uses a mobile phone connected to a laptop which translates voice commands in robot movements. The robot is connected through Ethernet network to the computer in order to receive orders. Another approach is the remote control based on an IP network, where the WWW is used to create a server/client model [3]. The user connects to a server through the internet and sends and receives commands to/from the robot. Robot control using SMS (Short Message Service) is also possible. An SMS message is sent to an external server that connects to the IP network and sends the control signal to the remote robot over the IP network. The SMS defines a special protocol which is analyzed by the server prior sending the control signal to the robot. There is also the possibility of using the DTMF (Dual Tone Multiple Frequency) generated when a keypad button of the mobile is pressed. The robot makes a call to the user mobile phone and its movements are controlled by sending the DTMF tone [2].

Current phones are equipped with wireless adapters that avoids the use of an external entity in order to control the robots. This phones run advanced operating systems like Android or iOS which allows the development and installation of diverse applications to extend phone functionalities (such as robots control applications). This so called smartphones, incorporates WiFi and Bluetooth adapters which allows the user to communicate with outside devices that implements such protocols.

Due to this fact, several robotics companies have launched new robot series which can communicate directly with smartphones using Bluetooth or WiFi protocols. Some examples are the iRobot Ava, a phone-controlled to monitor senior people, or the Lego MINDSTORMS NXT, a programmable line of robotic toys [7].

In this thesis, a new robot control application, called DroidStorm, is proposed. This application uses an Android-Enabled smartphone to control the movements of a set of Bluetooth-enabled Lego MINDSTORMS NXT robots built in tribot configuration. Moreover, the application is capable of creating and controlling collaborative robotic systems which work together to arrive to a common objective.

The application provides users with a new interface for controlling robots, which allows the use of the phone as a joystick, thanks to the orientation sensors the smartphones include. The robots can also be controlled by a joypad-like set of buttons, which the user can use to send basic orders to the robot as move forwards, move backwards, turn left or turn right. Movement orders are sent to the robot using the Serial Port Profile (SPP) of the Bluetooth specification [8].

It provides two main operating modes: **synchronization mode** and **follower mode**. In synchronization mode, the phone controls up to seven connected robots, sending the same movement commands to all of them.

In follower mode, a leader/follower approach is established. One of the robots becomes the leader and the rest become followers. The leader robot is the only one controlled by the phone and the followers follow leader's path. For the leader, a modulated infrared emitter has been specially built for that purpose. This infrared emitter is used as a point of reference for the followers. The followers use an infrared sensor capable of filtering modulated signals and has been programmed in order to use data received by the sensor to be able to follow the leader.

In addition, the application offers the possibility of recording movements. The user can perform several movements with the robots and record them in order to reproduce them later. This mode allows to use a robot as, for example, surveillance robot, covering one space just once and making it repeat it as many times as wanted without user interaction.

Furthermore, a predefined sets of movements from XML files can be used. This XML files can be created in external devices (such as computers) and loaded later in the application to make the robot move as defined. Both records and XMLs are loaded in a so called **Demo mode**.

Our proposed system employs Bluetooth, a versatile and flexible short-range wireless networking technology with low power consumption. Apart from the application, we developed a whole new Bluetooth library for Android as well as the logic and hardware behind the follower implementation.

This prototype system allows us not only to confirm the correct behavior of the designed application, but also to demonstrate the capabilities of mobile systems to control autonomous systems which uses wireless communications to establish intelligent communication spaces as well as collaborative robotic systems. All the developed code is freely available at <http://www.grc.upv.es>

1.1 MOTIVATION AND OBJECTIVES

The main motivation of this thesis is to develop a mobile application capable of controlling autonomous systems, using both user input or predefined sets of movements. Also, this application have to be capable of creating and controlling collaborative robotic systems. We are also particularly interested in knowing if mobile devices are suitable for controlling autonomous systems and evaluate whether or not they can be a candidate solution for such applications.

The main sub-objectives are:

- **Bluetooth library development:** A new Bluetooth library has to be developed in order to extend the application to all Android-Enabled devices regardless of the Android operating system version.
- **Lego protocol implementation:** A low level protocol has to be implemented in order to be able to communicate with Lego MINDSTORMS NXT robots.

- **“Leader lost” protocol:** A suitable protocol to control when the follower has lost the leader has to be developed.
- **Modulated IR Emitter building:** A modulated IR Emitter for Lego MINDSTORMS NXT has to be built in order to emit modulated infrared light. This IR emitter has to be controlled by the application remotely.
- **Follower programming:** The follower robot has to be programmed in order to follow leader's path using a infrared sensor.
- **Native application:** An Android based application, capable of controlling the robots from different inputs (user input, XML input) has to be developed.
It has to be able to create and control a collaborative robotic system where two robots work together to arrive to the same target. For this, it has to handle leader loose. Also, it has to be able to record user inputs and reproduce them later on.

The rest of this document is organized as follows: Chapters 2 and 3 describes the system architecture and presents the hardware which composes the system. Chapter 4 details the application architecture and its implementation. Chapter 5 describes the how the interaction between the smartphone and the robot is made. Chapter 6 describes the robot programming. Chapter 7 illustrates the evaluation of the proposal and finally, in chapter 8, some concluding remarks are given.

2 SYSTEM ARCHITECTURE

The overall system architecture is based on a master/slave model where an Android-Enabled phone acts as the master and can control up to seven Lego MINDSTORMS NXT robots which act as slaves. The phone (master) connects to the robots (slaves) and control their movements depending on an input that may be received from three different sources:

- **User input**
 - The user has the possibility to use two interfaces to move the robot.
 1. Robots may be moved using the phone in landscape mode as a joystick (control by movement). Android-Enabled devices are equipped with movement sensors which are used by the application to translate phone movements into robot movements.
 2. The user may also move the robot using buttons representing a joy-pad (control by buttons). Each button sends basic orders to the robot: move forwards, move backwards, turn left , turn right and stop.
- **Recorded movements**
 - User can record all movements made in user input modes and reproduce them later as many times as wanted. In this way, the user can easily create a predefined path for the robot just making it once.
- **XML input**
 - Predefined movements can be loaded from a XML file. This file must follow a well-defined structure which contains the parameters to be applied to the motors of the robot.

The phone is also capable of creating a collaborative robotic system, where two robots collaborate together to achieve a single objective. In this system, one of the robots is controlled by the phone and the other one follows it autonomously. In this way, one of the robots guide the second one to achieve a given objective. For its implementation, a IR emitter that we developed has been installed in guide robot's back, in order to allow the other one to follow its path by means of an IR sensor.

Figure 1 shows the overall system architecture.

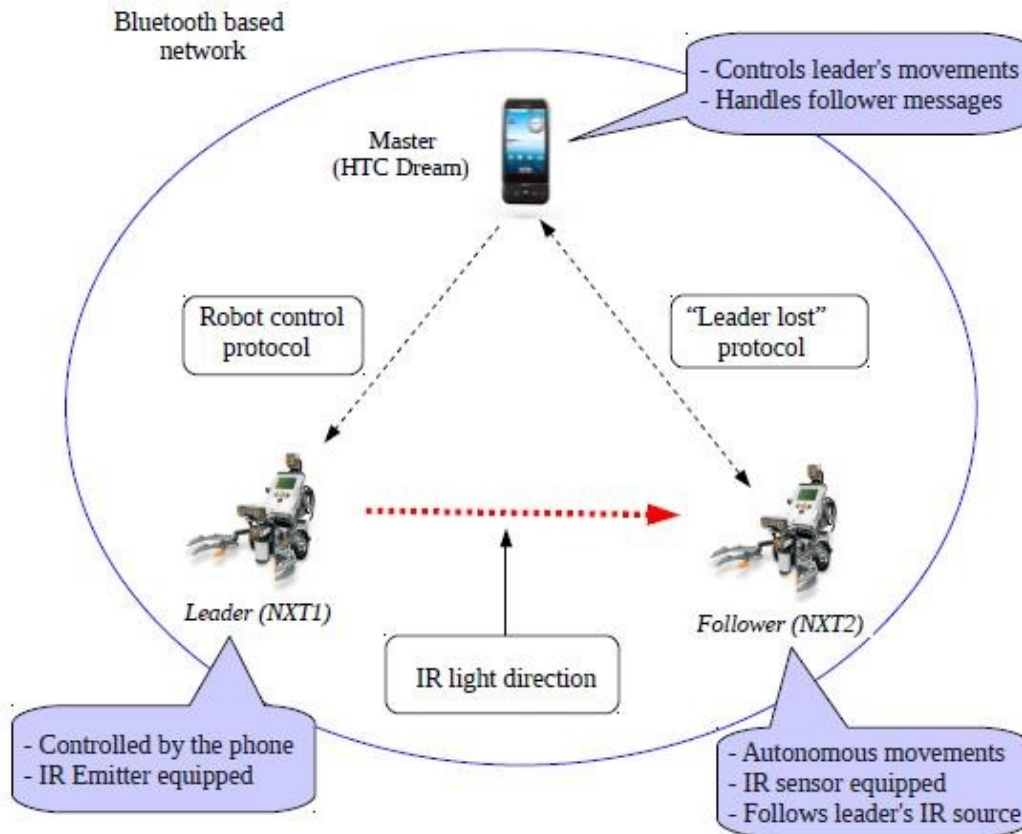


Figure 1: System architecture.

The phone uses the Bluetooth Serial Port Profile (SPP) to connect to the robots. Serial Port profile is one of the Bluetooth specification profiles that emulates a serial cable to provide a wireless substitute for existing RS-232 standard. SPP is based on the RFCOMM protocol, a simple set of transport protocols, made on top of the L2CAP protocol, providing emulated RS-232 serial ports. Lego MINDSTORMS NXT robots use this profile and the LEGO MINDSTORMS NXT Communication protocol (also known as Lego FANTOM protocol [9]) to receive commands from outside Bluetooth-Enabled devices such as mobile phones or computers.

This application has been developed using a HTC Dream mobile phone with Android 1.6 installed on it. Due to Bluetooth implementation wasn't introduced into Android until version 2.0 [15], a new Bluetooth library has been developed from scratch.

This new library also guarantees that this application will work in all Android-Enabled devices, regardless of its Android version. This library has been divided into two layers: a native layer that has been written in C and uses the Linux Bluetooth stack implementation (BlueZ), and a java layer, which, by means of JNI, will make calls to the native layer. This library will be introduced in detail in chapter 4.

The application has two operation modes. The default mode is **synchronization mode**, where the phone sends the same movement orders to all the robots it controls.

The second mode is called **follower mode**. This mode is a functional extension of synchronization mode and enables a second robot to move autonomously following the robot controlled by the phone. This mode can be enabled when the phone is connected to more than one robot. In this mode a leader/follower relationship is established between two robots: the robot controlled by the phone adopts a leader role, and the one following adopts a follower role.

The phone may also receive commands from the follower. If the follower loses the leader, it sends a message to the phone indicating which was the last direction where the follower was seen. In case that the robot is being controlled by the user, robot control passes to the application until the leader is found again. Depending on the direction, the phone will move the leader in such a way that the follower can find it. For this purpose, a communication protocol called "Leader lost" protocol has been created.

As said before, in follower mode, only one robot is controlled by the phone while the second follows it. Leader tracking has been implemented using a IR follower model which we have developed (both hardware and software): the leader has a specially built IR emitter installed in its back, and the follower uses a IR sensor to follow it.

With the purpose of minimizing external sources of noise (sunlight, etc), the IR signal has to be modulated, and the sensor has to be able to filter IR signals to keep only the modulated ones.

The leader has a modulated IR emitter specially built for the purpose, because there are no available IR emitters for LEGO MINDSTORMS at the moment. This emitter has been built to emit 600hz IR signals. The follower uses the IR Seeker V2 from HITECHNICS. IR Seeker V2 is an enhanced IR sensor which detects 600hz and 1200hz modulated signals. Also, is capable of returning the direction where the IR beacon is located.

IR Seeker V2 and home-made IR Emitter will be introduced with more detail in next chapter.

3 HARDWARE ENTITIES

Following, all hardware entities used for the development of this thesis will be introduced. This hardware entities includes the phone used for development (HTC Dream), the Lego MINDSTORMS NXT, the IR emitter, and the IR Seeker V2.

3.1 HTC DREAM



Figure 2: HTC Dream.

The Android-Enabled phone used to develop this thesis is the HTC Dream [8].

The HTC Dream (also marketed as T-Mobile G1 in the US and parts of Europe) was the first phone to the market to use the Android mobile device platform. The phone is part of an open standards effort of the Open Handset Alliance. The phone was first released in in the US on 22 October 2008 and in the UK on 30 October 2008 and arrived to the rest of Europe during 2009. In Spain, Telefonica also launched a slightly modified version of the phone (control buttons were modified) on 20 April 2009. Here is a summary list of hardware specifications for HTC Dream.

Processor	528 MHz Qualcomm MSM7201A ARM11 processor
Connectivity	- Wi-Fi (802.11b/g),
Memory	192 MB RAM
Display	320 x 480 px, 3.2 in (81 mm), HVGA, 65,536 color LCD at 180 pixels per inch (ppi)
Storage capacity	Flash memory: 256 MB - microSD slot: supports up to 16 GB
Input	capacitive touchscreen display, QWERTY keyboard, trackball, volume controls, 3-axis accelerometer
Camera	3.2 megapixel with auto focus
Power	3.7 V 1150 mAh Internal rechargeable removable lithium-ion battery
Dimensions	117.7 mm (4.63 in) (h) 55.7 mm (2.19 in) (w) 17.1 mm (0.67 in) (d)
Weight	158 g (5.6 oz)

Table 1: HTC Dream hardware specifications.

The HTC Dream has received official Android OS updates from version 1.0 to version 1.6, however since on July 27, 2010 it was officially discontinued. Despite of its discontinuity, is still possible to install later versions of Android OS by installing custom ROMs, but this requires the phone to be "rooted". When the device is rooted, it gives full access to the internal files of the phone, in particular, it allows changing and re-flashing the bootloader and operating system, which means that a totally custom ROM can be installed. One popular unofficial firmwares site is XDA-Developers [14] where independent developers work together to port newer Android versions to HTC Dream and other devices.

The Android version used in the thesis is the last official Android version released for the phone: Android 1.6 codename Donut. The application uses special features of this so called "smartphones on steroids", like the 3-axis accelerometer which is used to translate phone movement into robot movement.

3.2 LEGO MINDSTORMS NXT



Figure 3: Lego MINDSTORMS NXT: Tribot configuration.

LEGO MINDSTORMS is a line of programmable robotics/construction toys, manufactured by the LEGO Group. It comes in a kit containing many pieces including sensors and cables.

LEGO MINDSTORMS originated from the programmable sensor blocks used in the line of educational toys. The first retail version of LEGO MINDSTORMS was released in 1998 and marketed commercially as the Robotics Invention System (RIS).

The next version was released in 2006 as LEGO MINDSTORMS NXT. The newest version, released on August 5, 2009, is known as LEGO MINDSTORMS NXT 2.0.

The NXT version kit comes with three servo motors and one sensor each for light, sound, and distance as well as 1 touch sensor. The NXT 2.0 kit comes with 2 touch sensors as well as light, sound and distance sensors, and it supports up to 4 sensors without using a multiplexor. LEGO MINDSTORMS may be used to build a model of an embedded system with computer-controlled electromechanical parts. Many kinds of real-life embedded systems, from elevator controllers to industrial robots, may be modeled using LEGO MINDSTORMS.

The main component in the kit is a brick-shaped computer called the NXT Intelligent Brick. Here is a summary list of hardware specifications for the NXT brick.

Main processor	Atmel® 32-bit ARM® processor AT91SAM7S256 - 256 KB FLASH - 64 KB RAM - 48 MHz
Co-processor	Atmel® 8-bit AVR processor, ATmega48 - 4 KB FLASH - 512 Byte RAM - 8 MHz
Bluetooth wireless communication	CSR BlueCoreTM 4 v2.0 +EDR System - Supporting the Serial Port Profile (SPP) - Internal 47 KByte RAM - External 8 MBit FLASH - 26 MHz
USB 2.0 communication	Full speed port (12 Mbit/s)
4 input ports	6-wire interface supporting both digital and analog interface - Mainly used for sensors connection - 1 high speed port, IEC 61158 Type 4/EN 50170 compliant
3 output ports	6-wire interface supporting input from encoders - Mainly used for motors connection
Display	100 x 64 pixel LCD black & white graphical display - View area: 26 X 40.6 mm
4 button user-interface	Rubber buttons - Used to navigate a user interface using hierarchical menus
Power source	6 AA batteries - Alkaline batteries are recommended - Rechargeable Lithium-Ion battery 1400 mA is available
Connector	6-wire industry-standard connector - RJ12 Right side adjustment

Table 2: LEGO MINDSTORMS NXT Hardware specifications.

3.2.1 PROGRAMMING

Lego MINDSTORMS NXT comes bundled with the NXT-G programming language which uses a command box programming, rather than code programming. This means that rather than requiring users to write lines of code, they instead can use flowchart like "blocks" to design their program. All components (sensors, motors), are represented as this blocks and the data flow is represented as wires which connects each block. Despite of being really easy to use, it is very limited because of its limited complexity, and moreover, it shows huge lack of performance. Due to this limitations, and thanks to open specifications of Lego MINDSTORMS, other third-party programming languages have appeared.

This programming languages normally require to flash a special firmware with the purpose of enhancing Brick's capabilities. Some popular third-party languages are:

- C and C++, under brickOS, formerly LegOS
- C and Assembly, under the GCC open source firmware kit NXTGCC
- Java, under leJOS or TinyVM
- Not eXactly C, an open source C-like high-level programming language
- Not Quite C (NQC)
- RobotC

Also, the NXT Brick can be remotely controlled by Bluetooth or USB using the LEGO MINDSTORM Communication Protocol.

This protocol provides a simple interface for outside devices to use basic robot functionality without the need to write or run specialized remote control programs on the robot.

For this thesis, two tribots has been built as leader and follower. A tribot robot consists of two drive wheels and a trailing coast wheel and can be made to turn by driving the two drive wheels at different rates. In order to control the leader, the LEGO MINDSTORMS Communication Protocol will be used, whereas the logic behind the IR tracking of the follower will be implemented using RobotC.

3.3 NXT MODULATED IR EMITTER

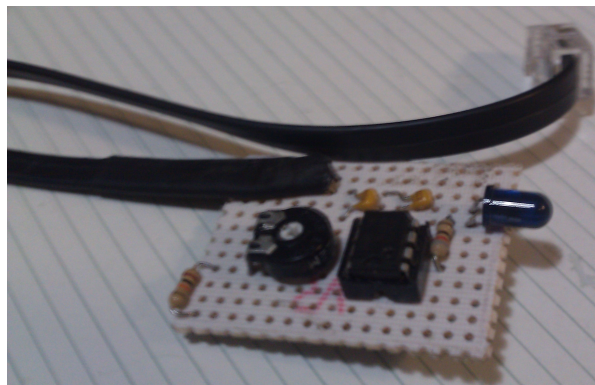


Figure 4: NXT Modulated IR Emitter

To achieve the maximum signal reception for the follower, the signal has to be free of external noise. Since no IR Modulated emitters are available for NXT MINDSTORMS at the moment, a new one has been built. The core of this emitter is a 8-pin 555 Integrated circuit which is used in astable mode to produce a 600hz square wave. A 5mm IR Led is connected to the circuit output to make it flash at that frequency. The whole circuit is connected to a NXT output for motors, so it can be easily turned on and off remotely using LEGO MINDSTORMS Communication protocol (as if it was a motor).

The result is a 600hz modulated IR signal installed in the leader, which can be filtered by the IR receptor installed in the follower. The following figure shows the schematic of the circuit used to build the IR Emitter.

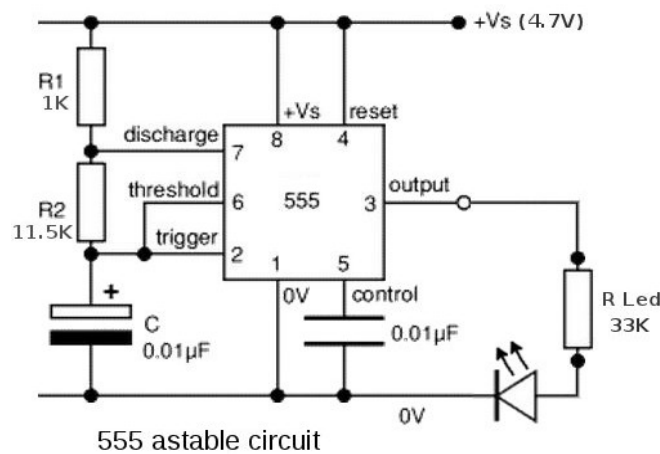


Figure 5: Circuit diagram

3.3.1 CIRCUIT MAIN COMPONENTS

THE 555 INTEGRATED CIRCUIT

The 555 timer IC was first introduced around 1971 by the Signetics Corporation as the SE555/NE555 and was also the very first and only commercial timer IC available. It provided circuit designers with a relatively cheap and stable integrated circuit for both monostable and astable applications. Although these days the CMOS version of this IC, like the Motorola MC1455, is mostly used, the regular type is still available, however there have been many improvements and variations in the circuitry. But all types are pin-for-pin plug compatible. In this thesis, the NE555 is the one used to build an astable circuit [16].

An astable circuit produces a 'square wave', this is a digital waveform with sharp transitions between low (0V) and high (+Vs). The circuit is called an astable because it is not stable in any state: the output is continually changing between 'low' and 'high'.

With the output high (+Vs) the capacitor C1 is charged by current flowing through R1 and R2. The threshold and trigger inputs, monitor the capacitor voltage and when it reaches 2/3Vs (threshold voltage) the output becomes low and the discharge pin is connected to 0V.

The capacitor now discharges with current flowing through R2 into the discharge pin. When the voltage falls to 1/3Vs (trigger voltage) the output becomes high again and the discharge pin is disconnected, allowing the capacitor to start charging again. This cycle repeats continuously unless the reset input is connected to 0V which forces the output low while reset is 0V.

An astable can be used to provide the clock signal for circuits such as counters.

RESISTORS AND CAPACITORS

To work as a astable, the 555 IC needs different resistors and capacitors. Depending on this values, the frequency of the output signal will be different. Following, the resistors and capacitors values needed to get the 600hz signal will be calculated.

The 555 astable cycle time (or time period) is the the time the square wave needs to complete one cycle. This cycle time (high time -Th- and low time -Tl-) is given by the following formulas

$$Th = 0,693(R1 + R2) * C$$

$$Tl = 0,693 * R2 * C$$

where R is in Ohms and C is in Farads. Having this two formulas we have that the complete cycle time is

$$T_{cycle} = Th + Tl = 0,693*(R1 + 2*R2)*C$$

since $F = 1/T$, we should use the following formula in order to get the frequency

$$F = 1 / 0,693*(R1 + 2*R2)*C$$

555 astable frequencies			
C1	R2 = 10kΩ R1 = 1kΩ	R2 = 100kΩ R1 = 10kΩ	R2 = 1MΩ R1 = 100kΩ
0.001μF	68kHz	6.8kHz	680Hz
0.01μF	6.8kHz	680Hz	68Hz
0.1μF	680Hz	68Hz	6.8Hz
1μF	68Hz	6.8Hz	0.68Hz
10μF	6.8Hz	0.68Hz (41 per min.)	0.068Hz (4 per min.)

Table 3: 555 astable typical values.

So now, we have to find correct values for capacitor C and resistors R1 and R2 to get the 600Hz output frequency we are looking for. The first value that should be chosen is the capacitor value, since this value will determine the output frequency range. According to the 555 typical values table, we should use a 0,1 microfarads capacitor, because the output range that it defines is from 6,8Hz to 680Hz.

Also, from this table, we get that the R1 should be 1KOhm, so the remaining value it's R2. If we develop the frequency formula, we get that R2 value is 11.5 Ohms

3.3.2 INFRARED LED

The IR Led chosen has been the 5mm Vishay TSUS5400 IR emitter [9]. This is a low cost infrared emitter with low forward voltage, which makes it suitable for this purpose, since in this way the NXT's power source lasts longer. Moreover, it's a 22 degree viewing angle led diode, which offers good visibility for the follower.

Every led diode should be used with a resistor in order to limit the current that flows through it. Excessive current will decrease its useful life, and can reduce the output of the LED substantially. In worst-case scenario, the led will overheat enough to burn out.

To get the led resistor, the following formula is used

$$R_s = (V_s - V_l) / I$$

where R_s is the resistor value in Ohms, V_s is the power supply voltage in Volts, V_l is the led voltage in Volts, and I is the led current in Amps. As said before, the circuit will be connected to a output port of the NXT, which offers 7,2 volts, but, due to the 555 IC, it drops to 5,5V. According to the TSUS5400 datasheet, the led voltage is 1,3V and the led current is 150ma.

Applying this values to the formula we get

$$R_s = (5,5 - 1,3) / (150 * 10^{-3}) = 25,33 \text{ Ohms.}$$

which is the value of the resistor that should be put between the output of the 555 IC and the IR diode. Due to, 25,33 Ohms is not a standard value for a resistor, a 33 Ohm will be used.

3.4 HITECHNIC IR SEEKER V2



Figure 7: IR Seeker V2 for Lego Mindstorms NXT.

The IR Seeker V2 is the IR sensor chosen to install in the follower robot [10]. It is a IR sensor for LEGO MINDSTORMS which has 240 degrees view. It returns the direction and the strength of the signal, making it perfect for this thesis purposes.

Also, it filters out background signals, like brightly lights or sunlight. It operates in 2 selectable modes:

- **Modulated (AC) Mode:** The sensor will detect modulated IR signals such as some IR remote controls. In Modulated mode the sensor will filter out most other IR signals to decrease interference from lights and sunshine for example. The sensor can be tuned to detect square wave signals at 600Hz and 1200Hz.
- **Un-modulated (DC) Mode:** The sensor will detect un-modulated IR signals such as sunlight.

The IRSeeker V2 detects the signal by using an array of 5 IR detectors. The signal direction returned is a value that represents a direction zone where the IR source is detected. This values range from 0 to 9, where 5 indicates that target is directly ahead, 1 indicates that the infrared target is left and behind, and 9 that the target is to the right and behind. A value of 0 is returned if no signal is detected, and -1 if there has been an error.

According to HITECHNICS documentation, zones are symmetrical areas around the sensor, but this is not really true. Zones 1, 3, 5, 7 and 9 are calculated using one sensor, while zones 2, 4, 6 and 8 are calculated by the interpolation of the sensors of the adjacent zones.

Direction value	Sensors activated (0-based)
1	Sensor 0
2	Sensor 0, Sensor 1
3	Sensor 1
4	Sensor 1, Sensor 2
5	Sensor 2
6	Sensor 2, Sensor 3
7	Sensor 3
8	Sensor 3, Sensor 4
9	Sensor 4

Table 4: IR Seeker V2: sensors activated depending on the direction.

As seen in the following graph, this approach causes that zones calculated by sensor interpolation are much narrower than the ones calculated by only one sensor [11].

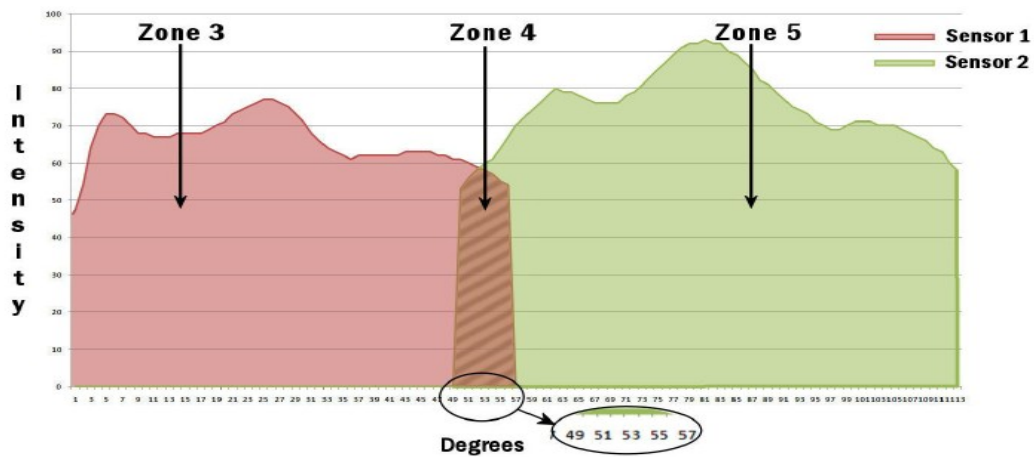


Figure 8: Relation between zone intensity and zone degrees

So a more accurate conceptual depiction of the zones for the IRSeekerV2 sensor is presented here.

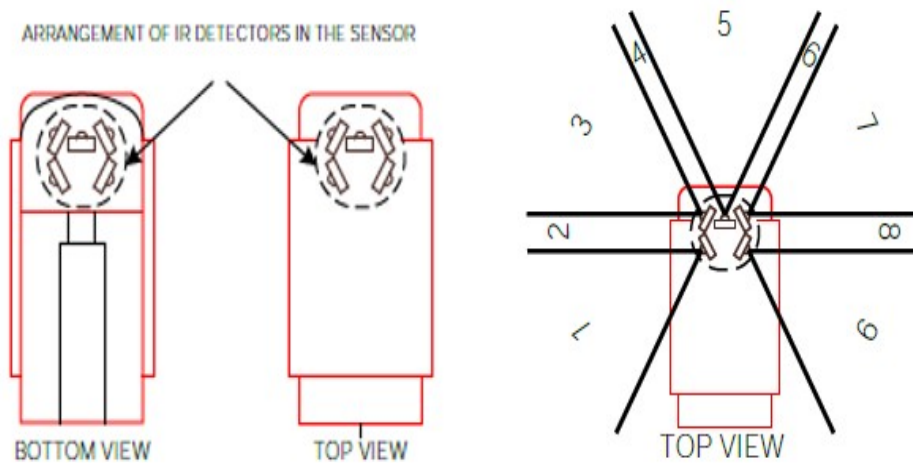


Figure 9: IR Seeker V2 sensor zones

For the follower implementation, the sensor has been mounted in such a way that the leader will be considered in front if the returned value is zone 5. Zones 4 and 6 will be used to know when the leader has started turning.

4 APPLICATION ARCHITECTURE

In this chapter, the application architecture and its implementation will be presented in detail. In order to control the robots, the application uses the proprietary Lego MINDSTORMS NXT Communication protocol which is already implemented in all MINDSTORMS robots firmware.

The Android implementation will be presented using a bottom-up approach, from the low level layer of the application (communication library and Lego protocol implementation) to the high level (Application activities). Bluetooth library implementation will be shown, because, as explained before, it had to be developed due to lack of implementation in the Android version installed in the developing device. Next, Lego Communication protocol will be explained and its implementation for Android will be shown.

Once the low layer of the application has been explained, all the activities and the logic behind them will be detailed with screenshots.

The Android implementation has two basic operating modes: **Synchronization mode** and **Follower mode**. In the synchronization mode, all movement orders are sent to all connected in sequential order, so all the devices will reproduce the same action at the same time. Despite of the delay introduced by Bluetooth communications, there is no significative difference between the first robot which receives the order and the last one.

In **Follower mode**, the phone drives one of the robots and a second robot (follower) follows the first one (leader) by means of a 600 Hz modulated IR beacon installed in the back of the leader. The follower will use a IR receptor capable of filtering IR signals in order to keep only those modulated at 600hz.

The application is intended to all Android-Enabled devices regardless of the Android version they are running, but it's important to remark that currently, this application doesn't work in HTC devices and Motorola devices. The reason is that both HTC and Motorola devices has broken BlueZ libraries, due to their UI customization (Sense and Motoblur respectively). It looks like this customizations had conflicts with some BlueZ libraries and, because of that, SPP profile is unavailable [29]. Therefore, in order to use this application in such devices, a custom Android ROM without Sense or Motoblur customization must be installed. This application has been tested successfully in HTC Dream phone running Android Donut 1.6 (currently outdated), and in HTC Desire HD phone running CyanogenMod 7, a custom ROM which includes Android Gingerbread 2.3 (the newest Android version when this thesis was written).

Application architecture can be divided in two main layers. A low layer, which contains the **Bluetooth library** and the **Lego MINDSTORMS NXT Communication protocol implementation**, and the high layer, which includes all the UI and the logic behind the robot control. The high layer is composed mainly by **handlers** and **controllers**. Handlers process the input (from user or storage), and passes the results to the controllers, which implements all the necessary logic to move the robot. All handlers and controllers implemented in this application will be detailed in next chapter.

Figure 11 shows the application architecture.

Application architecture diagram

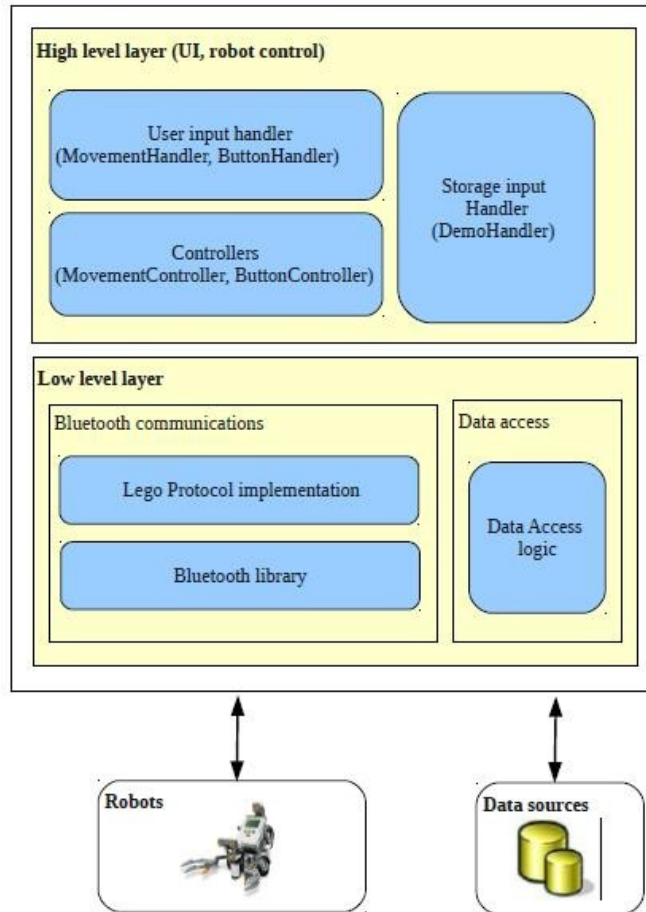


Figure 10: Droidstorm: Application architecture

4.1 LOW LEVEL LAYER: COMMUNICATIONS IMPLEMENTATION

4.1.1 BLUETOOTH LIBRARY (BTMANAGER)

Because of the Bluetooth implementation has not been introduced in Android's application layer until version 2.0, a Bluetooth library has been developed with the purpose of making the application work with the selected Android-Based phone. This library is capable of turning on and off the Bluetooth radio (Bluetooth control), and make connections and disconnections. The Bluetooth control function uses the Android Bluetooth API in case the phone is running Android version 2.0 or higher. Otherwise, the library uses the java libraries the operating system uses to control the Bluetooth. It instantiates these libraries using a reflection technique and uses its functions in order to control the Bluetooth adapter [22].

BTManager library is implemented using the Java Native Interface framework (JNI). The JNI framework enables a Java application to call and to be called by native applications.

It is divided in two layers: a native C coded layer, and a Java layer. The native layer uses the Bluetooth stack implementation for Linux called BlueZ. It is in charge of interfacing directly with the Bluetooth chipset at low level.

It makes the connections and disconnections, sends and receives data from Bluetooth. The Java layer acts as a wrapper for this native layer by means of JNI. This Java layer is used by the Android application in order to interact with the Bluetooth radio. Figure 12 shows the Bluetooth library architecture diagram.

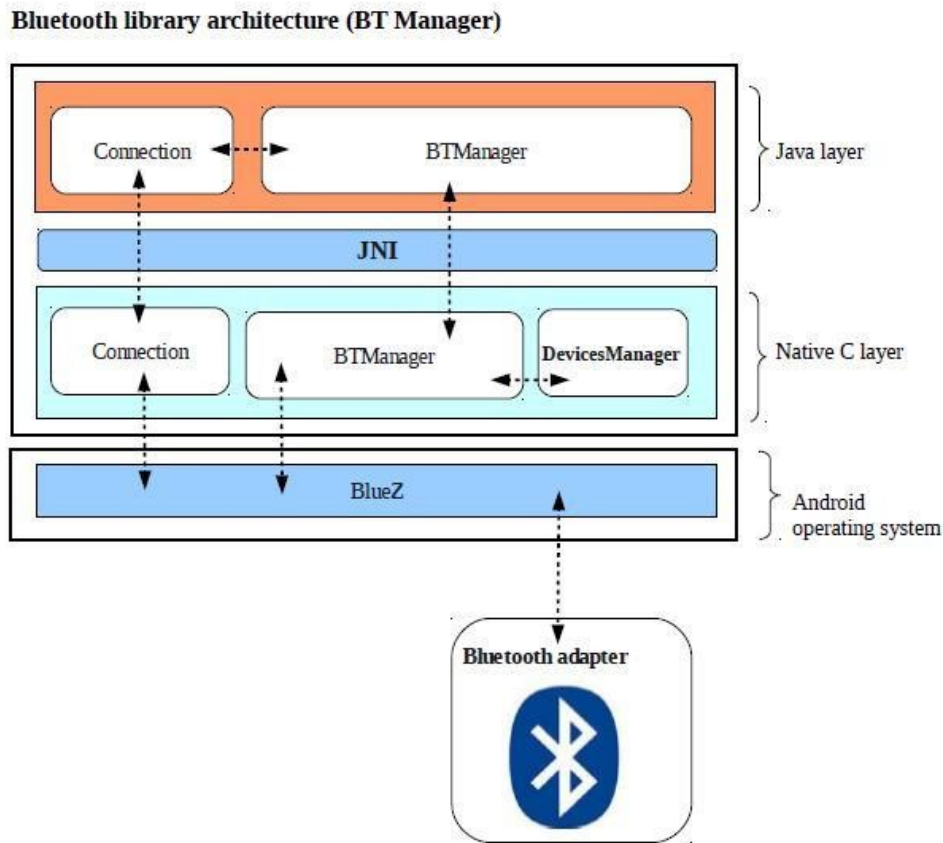


Figure 11: Bluetooth library architecture diagram

■ The native layer

The native library manages connection, disconnections and message sending at low level. It also maintains a relationships between Bluetooth addresses and opened sockets, in such a way that a possible developer using this library just have to take care of the Bluetooth addresses of the devices he or she wants to interact with.

In order to enable a native library to call and to be called from a Java application, it must include the JNI headers. Also, native functions exported by the library must have an equal method in the Java layer (java native methods).

This Java methods will be marked as native, which tells the compiler that the implementation of that method has been made in native code.

The native functions must have a fixed signature format, which is formed by some JNI keywords as well as the classname and the method name of the Java method which calls it.

JNI Required signature

```
JNIEXPORT <javaReturnValue> JNICALL Java_<ClassName_methodName>
(JNIEnv *, jclass, <javaArguments>)
```

When a native java method is called, the execution in the native layer. In order to manipulate Java objects, all native methods receive a pointer to an struct (JNIEnv) which, in turn, contains a pointer to the Java Virtual Machine (JVM). This pointer includes all necessary functions to interact with the JVM and create and work with Java objects.

To use the native library, the Java code must load it first using the method *System.load* or *System.loadLibrary*, which receives the absolute path to the native library in the first case, or just the library name if it is already in the Java ClassPath. If the library is not loaded properly, a *UnsatisfiedLinkError* exception will be thrown when a native method is called.

The native library interacts with the Bluetooth adapter using the Linux Bluetooth implementation library called BlueZ. Thanks to Android is a Linux-based operating system, this libraries are available in all Android versions. The only drawback of the native implementation is that it is machine dependent, which means that the library has to be compiled for all the system architectures where the application is supposed to run. This application has been compiled for all systems architectures supported currently for Android, and they are all included in the same APK file, so that when the application is launched and the native library is about to be loaded, the Android operating system detects the system architecture and then loads the correct library file. Table 5 shows the most important files from this layer and the functions they expose.

File	Exposed functions	Summary
Devices Manager	addDevice	Adds a device and its socket to a linked list
	delDevice	removes a device and its socket to a linked list
BTManager	discoverDevices	Searches for visible devices
	Connect	Connects to a device
	disconnect	Disconnects from a device
Connection	broadcastCommand	Sends a message to all connected devices
	SendSinglecommand	Sends a command to a single device
	WaitForMessage	Waits for a message from the given Bluetooth address

Tabla 5: Bluetooth library: Exposed functions from native layer.

Following, this files and its functions are explained in detail. The functions include some code snippets to show the most important parts of the implementation.

It is noteworthy that this code snippets have been highly summarized since the complete source code would be too long to be included in this thesis.

1. DevicesManager

This file does not exposes any function to the Java layer, but it's important because it maintains in memory all the connected robots as well as the opened sockets, and makes a relationship between them. In this way, the developer just have to take care of the Bluetooth address of the devices and not of the opened sockets. The relationship is maintained by means of a linked list formed by descriptor structs.

Descriptor struct

```
typedef struct descriptor
{
    // Bluetooth address
    char*  bdaddr;
    // Socket
    int socket;
    int connected;

    struct descriptor *next;
    struct descriptor *before;
}deviceDescriptor;
```

This library exposes functions to add and remove device descriptors from the linked list. It also has functions to get the socket id associated to a Bluetooth address, as well as all the sockets id from all connected devices.

2. BTManager

This file manages device discovery, device connection and disconnection. It includes the Bluez headers in order to be able to interact with the Bluetooth radio.

Included BlueZ libraries

```
#include <jni.h>
#include <string.h>
#include <errno.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>
#include <bluetooth/rfcomm.h>
#include <sys/socket.h>
```

This file exposes the following functions

discoverDevices

Function signature

```
JNIEXPORT jobject JNICALL Java_net_kaisoz_droidstorm_bluetooth_BTManager_discoverDevices
(JNIEnv * , jclass)
```

This function inquiries about visible robots to the local Bluetooth adapter. The discovered devices are returned back to the java layer in a Java Map object created for the purpose in the native layer. It's important to know that the result is filtered, so only Lego MINDSTORM devices will be returned.

Inquiring devices with BlueZ

```
// Get local device ID
dev_id = hci_get_route(NULL);
// Open device
sock = hci_open_dev( dev_id );
if (dev_id < 0 || sock < 0) {
    LOGE("Error opening local Bluetooth device: %s", strerror(errno));
    throwBluetoothException(env, "Error opening local Bluetooth device");
}

// Flush previous discovery history
flags = IREQ_CACHE_FLUSH;
ii = (inquiry_info*)malloc(max_rsp * sizeof(inquiry_info));

// Search for devices
num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL, &ii, flags);
```

Connect

Function signature

```
JNIEXPORT jobject JNICALL Java_net_kaisoz_droidstorm_bluetooth_BTManager_connect
(JNIEnv *env, jclass obj, jobjectArray jbtAddresses)
```

Connects to all Bluetooth addresses passed as arguments. These addresses are passed in a Java String Array (jbtAddresses). In this way, the application can connect several devices at once. The connection result is passed back to the Java layer using a Java Map which contains two entries:

- **success:** contains a Java Array with the Bluetooth addresses of the robots connected successfully
- **error:** contains a Java Array with the addresses of the robots which couldn't get connected

This Map entries will be used by the java layer to show the connection result to the user.

The code to connect to a device with BlueZ is the following:

Connect to a device with BlueZ

```
// Open Bluetooth stream socket. Use the rfcomm as a transport layer
newsocket = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
// set the connection parameters (who to connect to)
addr.rc_family = AF_BLUETOOTH;
addr.rc_channel = (uint8_t) 1;
// Translate char address to a Bluetooth address struct
str2ba( bdAddr, &addr.rc_bdaddr);
// connect
status = connect(newsocket, (struct sockaddr *)&addr, sizeof(addr));
```

If the connection has been successful, both the address and the related socket will be added to the DevicesManager. Also, the Bluetooth address will be added to a Java Array which later will be linked to the **success** entry of the Java Map. Otherwise, the Bluetooth address won't be added to the DevicesManager, but it will be added to the Java Array used for the **error** entry in the Map.

Disconnect

Function signature

```
JNIEXPORT jobject JNICALL Java_net_kaisoz_droidstorm_bluetooth_BTManager_disconnect
(JNIEnv *env, jclass obj, jobjectArray jbtAddresses)
```

Disconnects from all Bluetooth addresses passed as argument. This argument is passed in a Java String Array (jbtAddresses). In this way, the application can disconnect several devices at once. The connection result is passed back to the Java layer using a Java Map which contains two entries:

- **success**: contains a Java Array with the Bluetooth addresses of the robots disconnected successfully.
- **error**: contains the addresses of the robots which couldn't get disconnected.

This Map entries will be used by the java layer to show the disconnection result to the user.

The code used to disconnect from a device with BlueZ is the following

Disconnect from a device with BlueZ

```
shutdown(socket, SHUT_RDWR);
close(socket);
```

If the disconnection has been successful, both the address and the related socket will be removed from the DevicesManager. Also, the Bluetooth address will be added to a Java Array which later will be linked to the **success** entry of the Java Map. Otherwise, the Bluetooth address won't be removed from the DevicesManager, but it will be added to the Java Array used for the **error** entry in a Map.

3. Connection

This file contains functions to translates java messages to native messages and the other way round. Also, sends and retrieves messages to/from the robots and listens to follower messages.

The most important functions exposed by this file are the following

broadcastCommand

Function signature

```
JNIEXPORT jcharArray JNICALL
Java_net_kaisoz_droidstorm_bluetooth_Connection_broadcastCommand
    (JNIEnv *env, jclass obj, jobject data, jboolean response)
```

This function is used in synchronization mode to send the same command to all the connected robots. This function receives a Java char array which contains the command to send to the robot and translates it to an unsigned char array.

This is done using a function called `translateToNative` contained in this same file. This step is needed because unsigned char is the suitable type of data to hold the bytes that will be sent to the robot.

The function also receives a boolean value to indicate if a robot response is needed. If this value is set to false, null value is returned. Otherwise, the function will return only if either receives a response from the robot or a timeout happens. If response is received from the robot, it will be translated to a Java Char Array using `translateToJava` function, and passed back to java layer.

Once it gets the translated message, it calls the `getSocketFromConnDevices` function from `DevicesManager` to get all the opened sockets. Then, iterates over all the sockets, and uses its value to send the command using the `doCommand` function This function will prepare the Bluetooth packet and will send it to the robot.

sendSingleCommand

Function signature

```
JNIEXPORT jcharArray JNICALL
Java_net_kaisoz_droidstorm_bluetooth_Connection_sendSingleCommand
    (JNIEnv *env, jclass obj, jstring jbtAddr, jobject data, jboolean response)
```

This function does the same as `broadcastCommand`, with the difference that this sends the command only to one robot. This function is used when some operations has to be done in just one robot, such program listing, or program starting.

It receives the robot Bluetooth address (passed in the `jbtAddr` variable as a java String), the command as a java char array and a boolean value, which indicates if the function should wait for robot response. As `broadcastCommand` function, the first step it takes is the translation of Java char array to unsigned char array. After this, it gets the socket associated to this Bluetooth address from `DevicesManager` using `getSocketByBTAddr` function and calls `doCommand` function to send the command.

doCommand

Function signature

```
int doCommand(int socket, int waitForResponse, unsigned char *command, int cmdLen,  
              unsigned char **response, int *respLen)
```

Although this function is not exposed to the java layer, it's important to mention it because is the one which sends the commands to the robots. This function receives a socket, a translated robot command and the command length. If waitForResponse is set to 1, it waits for the robot response until it arrives or a timeout happens.

If the response is received, the out response variable will be filled with the robot response and passed back to the calling function.

Send command to the robot

```
// create a unsigned char pointer to the write buffer  
unsigned char *wBuf = NULL;  
// Initialize the write buffer with size commandLength + 2 bytes for Bluetooth headers  
wBuf = (unsigned char*)malloc((cmdLen + 2)*sizeof(char));  
// Convert the command length to short int in order to put it into the Bluetooth headers  
short int si = (short int) cmdLen;  
// Fill the write buffer with command length and the command itself  
memcpy(&wBuf[0], &si, 2);  
memcpy(&wBuf[2], command, cmdLen);  
// Write it to the Bluetooth socket  
writtenB = write(socket, wBuf, cmdLen + 2);
```

Receive response from the robot

```
// Prepare file descriptor set and set the socket to listen to  
fd_set set;  
FD_ZERO(&set);  
FD_SET(socket,&set);  
  
// Prepare timeval struct for timeout.  
struct timeval timeout = {1,2};  
  
// create a unsigned char pointer to the read buffer  
unsigned char *rBuf = NULL;  
  
// Listen to robot response up to 1 second  
rtn = select(socket +1, &set, NULL, NULL, &timeout);  
// If some message has been received  
if(rtn > 0){  
    // read the first two bytes which will indicate the incoming message size  
    while ((readB = read(socket, &rBuf, 2)) == -1) {}  
    // Save message size in a variable  
    memcpy(&tamRet, &rBuf[0], 2);  
    // Initialize the read buffer with the size received  
    rBuf = (unsigned char*)malloc((tamRet)*sizeof(char));  
    // Read the rest of the message  
    while ((readB = read(socket, &rBuf, tamRet)) == -1) {}  
}else{  
    LOGE("No response recieved");  
}  
}
```

waitForMessage

Function signature

```
JNIEXPORT jcharArray JNICALL  
Java_net_kaisoz_droidstorm_bluetooth_Connection_waitForMessage  
(JNIEnv *env, jclass obj, jstring jBtAddr)
```

This function is used in follower mode to listen to follower messages. It receives the Bluetooth address to listen to, and waits for a message by an active waiting controlled by the Java layer. The function implementation is similar to the response receiving of doCommand function. Unlike doCommand, when the function receives the message, it returns the translated response directly to the Java layer.

■ **The Java layer**

The Java layer is the interface the application uses to interact with the Bluetooth adapter. This layer communicates with the native layer by means of JNI, which is a framework that enables a Java application to call and to be called by native applications. Since it is a highly used library, it has been implemented as a singleton in order to reduce the number of instances of objects in memory. In this way, the library objects are instanced only once and are available during all application life cycle. Table 5 shows the most important files from this layer and the functions they expose. Functions exposed in this table, does not include native functions, because, as explained before, they are just used to pass execution flow to the native layer.

File	Exposed functions	Summary
BTManager	initialize	Initialized Bluetooth adapter
	getConnection	Returns a connection object which abstracts a Bluetooth connection to a device
Connection	sendCommand	Sends a command to a robot
	waitForMessage	Waits for a follower message

Tabla 6: Bluetooth library: Exposed functions from Java layer.

1. BTManager

This file is the principal one in all the Java layer. It controls the Bluetooth adapter and acts as a wrapper for the BTManager native file.

Due to the lack of Bluetooth API implementation in Android 1.6, Bluetooth control function uses the java libraries that the operating system uses in order to control the Bluetooth adapter. It instantiates these libraries by means of reflection technique and uses its functions in order to control the Bluetooth adapter. If the application detects that it is running in a device with Android 2.0 or higher installed, it uses Android Bluetooth API. Bluetooth device can be turned on, off and can detect its current state no matter the Android version installed.

BTManager native methods definitions

```
public native IndexedMap connect(String[] bdAddresses);
public native IndexedMap disconnect(String[] bdAddresses);
public native IndexedMap discoverDevices() throws BluetoothException;
```

This file maintains in memory all the connected devices, associating its device name with its Bluetooth address. It exposes methods which returns the Bluetooth address associated to a name and the other way round.

The most important methods are explained below

initialize

Method signature

```
public BTManager initialize(Context context)
```

This method initializes the singleton, the Bluetooth objects needed to control the Bluetooth adapter, and loads the native library. If the singleton it's already initialized, it just returns the instance. Also, it registers a BroadcastReceiver to receive all Bluetooth events that may happen in the system, such as Bluetooth enabling or disabling from outside of the application. As said before, this application instances the correct Bluetooth objects files regardless of the Android version. It is done by reflection.

First, it tries to initialize the adapter by means of Android Bluetooth API. If this fails, means that the Android version installed is lower than 2.0, so operating system objects are used in order to get a reference to the Bluetooth adapter.

Android Bluetooth initialization

```
try {
    // Try Android 2.0 Bluetooth initialization
    Class<?> bluetoothAdapterClass = null;
    bluetoothAdapterClass = Class.forName("android.bluetooth.BluetoothAdapter");
    Method getAdapterMethod = bluetoothAdapterClass.getMethod("getDefaultAdapter",
(Class[]) null);
    mAndroid2 = (mDevice = getAdapterMethod.invoke(bluetoothAdapterClass, (Object[])
null)) != null;
} catch (Exception ex) {
    // If it fails, means that the Android version installed is lower than 2.0
    mAndroid2 = false;
}

if (!mAndroid2) {
    // If its not Android 2.0 or higher, get Bluetooth by means of a system service
    mDevice = mContext.getSystemService("bluetooth");
}

try {
    // Get enable, disabled and isEnabled methods. This should be done regardless of
the Android version installed
    if (mDevice != null) {
        Class<?> c = mDevice.getClass();
        mEnable = c.getMethod("enable");
    }
}
```

```

        mEnable.setAccessible(true);
        mDisable = c.getMethod("disable");
        mDisable.setAccessible(true);
        mIsEnabled = c.getMethod("isEnabled");
        mIsEnabled.setAccessible(true);
    }

    // Load the native library
    System.loadLibrary("BTCommunication");

} catch (Exception e) {
    e.printStackTrace();
    return null;
}

```

getConnection

Method signature

```

public Connection getConnection()
public Connection getConnection(String btAddress)

```

Returns a connection object to the specified Bluetooth address. In case that the no arguments method is used, it returns an connection object which will send commands to all connected devices.

2. Connection

This file acts as a wrapper for the *Connection* native file. It's constructor is defined as protected, so a Connection object can only be get by means of BTManager. Depending on how was created, the command will be send to all the robots or just to one robot

Connection object constructors

```

// Broadcast command constructor
protected Connection(){
    mode = MODE_BROADCAST;
}

// Send single command constructor
protected Connection(String btAddr){
    mode = MODE_SINGLE;
    this.btAddr = btAddr;
}

```

All native methods **are** marked as private, because previous operations have to be made before passing the execution to the native layer. For example, before sending a command, it checks the sending mode to know if the command should be send to all the connected robots or just to one robot. Therefore, for using the native methods, the public interface should be used.

```
public char[] sendCommand(char[] values, boolean response) throws BluetoothException
public char[] waitForMessage() throws BluetoothException
```

4.1.2 THE LEGO MINDSTORMS NXT COMMUNICATION PROTOCOL

Commands are sent to the robot using the Lego MINDSTORMS NXT Communication Protocol [33] (also known as Lego FANTOM), a proprietary protocol from Lego used to control MINDSTORMS robots from outside devices, such as mobile phones or computers. This protocol provides a simple interface for these outside devices to utilize basic robot functionality without the need to write or run specialized remote control programs on the robot. It makes possible to control the robot either through the USB communication channel or through the Bluetooth communication channel. It defines two types of telegrams: **Direct command telegrams** (ie: motor control, sensor readings, playing sounds..) and **System command telegrams** (system management: read files, write files, run programs...).

Figure 12 shows the general telegram architecture.

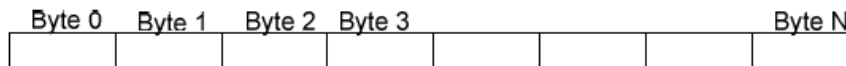


Figure 12: Lego MINDSTORMS NXT Communication protocol. Telegram architecture.

Byte 0: Telegram type

0x00: Direct command telegram. Response required

0x01: System command telegram. Response required

0x02: Reply telegram

0x80: Direct command telegram. No response

0x81: System command telegram. No response

Byte 1-N: The command itself or a reply, depending on telegram type. Bytes from N+1 will be ignored

The LEGO FANTOM protocol specification states that any incoming protocol telegram may be marked with the 0x80 mask on its telegram type byte to indicate that no response is expected. Direct commands is a primary use case for this functionality, as requiring a response on all telegrams could lead up to approximately 60ms latency.

All response packages include a status byte, where 0x00 means **success** and any non-zero value indicates a specific error condition.

All single byte values are unsigned, unless specifically stated. Internal data type is listed for all multi-byte values and all are assumed to be little-endian.

■ Maximum command length

Total direct command telegram size is limited to 64 bytes, including the telegram type byte. This limit doesn't include Bluetooth additional two bytes used for packet size. So the complete Bluetooth message will look like this

Length, LSB	Length, MSB	Command Type	Command	Byte 5	Byte 6	Etc.
-------------	-------------	--------------	---------	--------	--------	------

Figure 13: Lego MINDSTORMS NXT Communication protocol. Bluetooth packet.

SETOUPUTSTATE COMMAND

The most used command in the application is the *SetOutputState* command. This command is used to control the robot behavior: it allows the sender to control motors power, motors turn ratio and motor travel distance amongst other things. A fixed command configuration is used in the application.

The chosen configuration applies synchronization to two motors to make them go at the same speed. NXT firmware takes care of the speed of each motor, braking one of them if it goes faster than the other. Also, motor synchronization allows the application to control each motor turn ratio, with the purpose of making the robot turn as the user wants. To apply motor synchronization, the same command has to be sent two times, one for each motor. The first motor will be the master motor and the other one will be slave, which will be synchronized to the master. This forces the application to send two commands for each movement, one for each motor, although there is not impact in the application performance.

The chosen MINDSTORM configuration has been the tribot. A tribot robot consists of two drive wheels and a trailing coast wheel and can be made to turn by driving the two drive wheels at different rates. To achieve this with MINDSTORMS, it is needed to use motor synchronization and the turn ratio value. The synchronized wheel speed, will determine the turn direction: for example if the right wheel goes slower than the left wheel, the robot will turn left.

Depending on the turn ratio value, the NXT firmware sets different power to the synchronized wheel: the lower the turn ratio value is, the less power will be applied to the motor, so then, the sharper the curve will be. Travel distance can be also set in TachoLimit byte. This byte determines how many degrees the wheels will turn before stopping. In this application, the TachoLimit is set to 0 (no travel distance applied) because this is not known beforehand.

The SetOutputState command has the following fixed configuration

- **Byte 0:** 0x80. No response required
- **Byte 1:** 0x04. Command type (Motor Control)
- **byte 2:** Output port (Range 0-2)
- **byte 3:** Power byte (Range -100,100. Negative values means that the motor should turn backwards. Positive values make the motor to turn forward)
- **Byte 4:** 0x01 | 0x04 (Turn on the specified motor and turn on the regulation)
- **Byte 5:** 0x02 (Motor synchronization will be enabled. Needs enabled on two output)

- **Byte 6:** Turn Ratio (UnsignedByte. Range -100,100).
 - Only works when synchronization is enabled. Turn ratio of a motor regarding to the other one. Positive values will apply the turn ratio value to the slave motor. Negative values will apply the turn ratio values to the master motor, so it could be seen like role exchange.
 - A value of 0 means that both motors should turn at the same speed and the same direction as the master motor.
 - A value of 100 means that the slave motor should go at the same speed but in the opposite direction than the master direction.
 - A value of -100 means the same as 100 value, but exchanging directions: slave motor goes in the direction as the master motor should, and the master motor goes in the opposite direction.
- **Byte 7:** 0x20 (Output will be running)
- **Byte 8 -12:** 0 (No Travel distance applied)

Maximum message size: 13 bytes

■ Implementation

For the Lego FANTOM protocol implementation, two entities has been implemented: the command itself and the message. Short commands (3 bytes or less) has been implemented directly in a class (discussed later). Long commands (for example *SetOutputState*) has a message object associated (*MotorMessage*). A message object is a java bean object which contains all the values needed to perform the command. Not all the commands has been implemented, only a subset suitable for the developing of this thesis.

■ Message object implementation

In the Java Virtual Machine, bytes, shorts and ints are all four bytes long and are all signed. Hence, when two bytes are added together, a 32-bit arithmetic is performed. And when the result is stored back into a byte, the high 24 bits are not looped off, because the number is signed, and sign bit has to be retained. This same applies to the Dalvik Virtual Machine used in Android.

As said before, all values in Lego Fantom Protocol must be unsigned, so the chosen data type in Java to hold protocol values has been the char value, because it is the only Java data type that is unsigned. Since each Java char is 2 bytes long, each char will hold two bytes of a Lego message. This approach also reduces the data quantity passed to the native layer of the Bluetooth library, what is an expensive operation.

A message object contains a char array with size equal to half of the Lego message size rounded up. For example, In case of the *SetOutputState* command (13 bytes message), the *MotorMessage* object has a char array with size 7. Each message object has getter and setters for each command field, such as power field or turnRatio field. Each setter does the proper logical operations in order to fill correctly the char value which is meant to hold the byte. It takes into account if the value should be in the first or second byte of the char value.

Set the turn ratio in MotorMessage

```
**
 * Sets the turnRatio byte field
 * @param turnRatio int range from -100 to 100
 */
public void setTurnRatio(int turnRatio)
{
    // turnRatio and runState will share the same char
    // Put high part of the message char to 0
    values[3] = (char) (values[3] & 0x00FF);
    // Get the integer has a char
    char bTurnRatio = (char) (Integer.valueOf(turnRatio).byteValue() & 0xFF);
    // Left shift the turnRatio char in order to have it in the high part
    bTurnRatio = (char) (bTurnRatio << 8);
    // OR operation with the message byte to set it in the high part
    values[3] = (char) (values[3] | bTurnRatio);
}
```

The messages implemented as java objects are the following:

- **MotorMessage:** Message for *SetOutputState* message command. Contains all the values needed to set the behavior of the motors
- **MotorStateResponse:** Response from a *GetOutputState* command which contains the motor state. Returns the operation result and same values that are send with *SetOutputState* command, as well as the current distance traveled, number of motor counts since the last reset of the motor counter, current position relative to the last programmed movement and the current position relative to last reset of the rotation sensor.
- **GenericResponse:** Generic response from a Direct command. Contains the operation result as well as an error code in case of the operation was unsuccessful.
- **FindFileResponse:** Generic response from *FindFirst* and *FindNext* operations. These operations are used to list all the programs installed in the robot. Contains the operations result as well as a file handler and a file name.
- **CloseHandleResponse:** Generic response from *closeHandle* operation. Returns the operation result code as well as the handle that has been closed.

■ Command implementation

Commands has been implemented as functions grouped in classes depending on its purpose. For example, all the commands that act on motors, are grouped in a class called *MotorInterface*, and the commands related to program run and program list, has been grouped into a class called *MiscInterface*.

These classes extend a base class called *NXTInterface*. This class includes a *setConnection* method in order to set the connection object that will be used to perform the command. It also contains all needed constants for the direct command messages such as motor Ids or regulation mode.

This constants are hold also in java chars and have the correct byte order with the purpose of reducing the number of logical operations during the message construction.

The implemented interfaces are the following:

1. MotorInterface

This class groups all the commands used to control motor's behavior. The most important methods are the following:

setOutputState

Method signature

```
public GenericResponse setOutputState(MotorMessage message) throws BluetoothException
```

Sends a message to control a motor. It receives a *MotorMessage* object which contains the motor values to be set to the robot. Returns a *GenericResponse* in case the message requires a response from the robot.

resetMotorPosition

Method signature

```
public GenericResponse resetMotorPosition(char motor, boolean isrelative, char messageType) throws BluetoothException
```

Resets motor distance counter. It should be sent to each motor before performing a new movement, because otherwise, the robot returns to its start position before the next movement.

This function receives three values:

- *motor*: Motor ID constant. Motor IDs are specified in *NXTInterface*.
- *isrelative*: if true, the counter should be reset relative to motor last movement. If false, the counter will be reset to an absolute position.
- *messageType*: constant specified in *NXTInterface*. Determines if the message should return a robot response. In case it should, a *GenericResponse* object is returned.

getOutputState

Method signature

```
public MotorStateResponse getOutputState(char motor) throws BluetoothException
```

Gets the state for a given motor ID. Motor IDs are specified in *NXTInterface*. Returns a *MotorStateResponse* object.

2. MiscInterface

Groups all the commands related to program listing and execution. The most important functions are the following:

startProgram

Method signature

```
public GenericResponse startProgram(String name, char messageType) throws BluetoothException
```

Starts a program already stored in the robot. This method receives two arguments

- name: Name of the program.
- MessageType: constant specified in NXTInterface. Determines if the message should return a robot response. In case it should, a GenericResponse object is returned.

findFirst

Method signature

```
public FindFileResponse findFirst(String findIt) throws BluetoothException
```

Search for the the first file in robot storage which name contains *findIt*.

It returns a *FindFileResponse* object, composed by a handler id used to search for the next file using the same pattern, and the found file name. The application search files with “.rxe” extension, which are the executable programs installed in NXT.

findNext

Method signature

```
public FindFileResponse findNext(int handle) throws BluetoothException
```

Used in conjunction with *findFirst* method. It receives the handler returned by *findFirst* method to search for the next file which contains the same string as the one searched in *findFirst* function. It returns a *FindFileResponse* object, composed by the next handler id that should be used in next search, and the found file name.

This method should be called several times after *findFirst* until the result code is *FILE_NOT_FOUND*, which means that no more files match the specified pattern. Each opened handler must be closed with *closeHandler* method after it is not longer needed.

closeHandle

Method signature

```
public CloseHandleResponse closeHandle(int handle) throws BluetoothException
```

Closes an opened file handle in the robot. Receives the handleid. Returns a *CloseHandleResponse* object that contains the result code and the handler that has been closed.

Once introduced the most relevant methods used to control the robot, here is shown the source code that moves the robot.

SetOutputState command configuration from Java

```
motorInterface.resetMotorPosition(masterWheel, true,
MotorInterface.MESSAGE_TYPE_NO_RESPONSE);
motorInterface.resetMotorPosition(slaveWheel, true,
MotorInterface.MESSAGE_TYPE_NO_RESPONSE);

message.setTachoLimit(0L);
message.setMode(MotorInterface.MOTOR_MODE_ON_REGULATED_BRAKE);
message.setTurnRatio(turnRatio);
message.setMotorNum(masterWheel);
message.setPower(power);
```

```
message.setRegulationMode(MotorInterface.MOTOR_REGULATION_MOTORSYNC);
message.setRunState(MotorInterface.MOTOR_RUNSTATE_RUNNING);
motorInterface.setOutputState(message);
message.setMotorNum(slaveWheel);
message.setOutputState(message);
```

4.2 HIGH LEVEL LAYER: USER INTERFACE

After explaining the low level layer of the application, the high level (the one related with the user) will be explained. According to its functionality, the application can be divided into two modes: **synchronization mode** and **follower mode**. Synchronization mode is the default mode where the application controls all the robots in the same way. Follower mode is actually a functional extension of synchronization mode, because it just adds the possibility of changing the leader behavior according to the follower needs.

Android applications interact with the user by means of Activities. An Activity is one of the main components of an Android application and represents a single screen with a user interface. Normally, an Activity will be the entry point of an Android application.

Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. Activities works directed by callback methods that must be implemented. Android system will call this callback methods when the activity transitions between various states of its lifecycle, such as when the activity is being created, stopped, resumed or destroyed [15].

Activities are often presented to the user as full-screen windows, although they can also be used in other ways: as floating windows or embedded inside of another activity. The Activity class takes care of creating a window where Views can be placed. A View is a basic user interface element (like buttons or checkboxes) represented as a class, which handle screen layout and interaction with the user. Every view extends from the View class.

All different activities of the application will be exposed with some screenshots in order to make the explanation clearer. First, all activities related to synchronization mode will be explained, starting from the moment when the application is first launched. Finally, activities related with follower mode will be shown.

4.2.1 SYNCHRONIZATION MODE

The application starts with an **splash screen**, which initializes the *BTManager* and loads the native library. In case the Bluetooth radio is not enabled, the activity enables it. Also, it loads the language selected by the user (if any was selected previously). If no language was selected, it loads the default phone language.

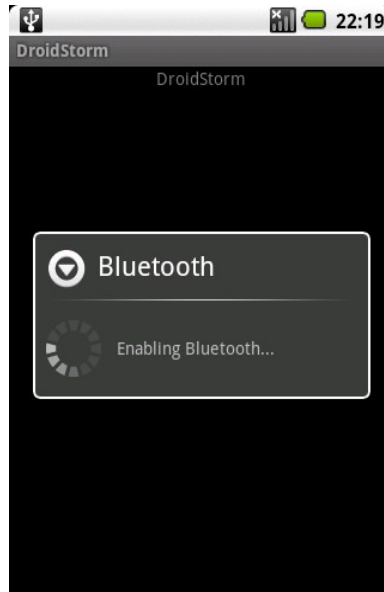


Figure 14: DroidStorm: Splash screen

Once everything has been initialized, the **ConnectionManagerActivity** appears. It presents three main buttons labeled as *Connect*, *Disconnect* and *Search*.

When the application launches, only the *Search* button is enabled, the connect and disconnect button will be disabled until some device is selected. When the user taps search button, a floating window appears informing the user that he has to wait for the searching.

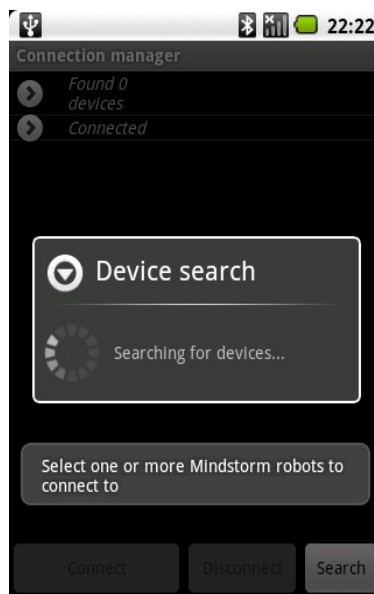


Figure 15:
ConnectionManagerActivity:
Device search

After searching, all MINDSTORMS devices found will be listed. Searching results are filtered so only MINDSTORMS devices will appear in the list.

Once the user has selected one or more MINDSTORMS devices, the *Connect* button is enabled. If the user taps the *Connect* button, the application will make the connection and show a pop up informing the result.

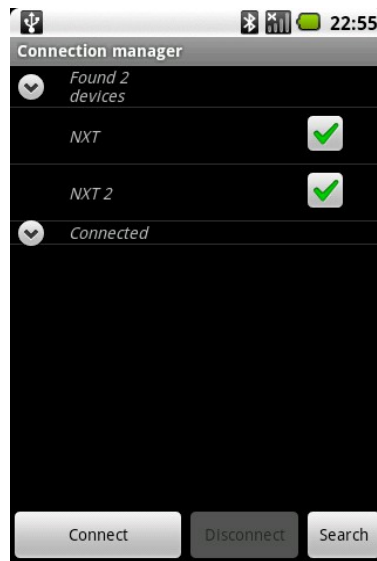


Figure 16:
ConnectionManagerActivity:
Found devices.

Once connected to one or more devices, the ***HandlerSelectorActivity*** is launched. This activity shows three different buttons to allow the user which kind of handler it want to use. Each handler is implemented by an activity.

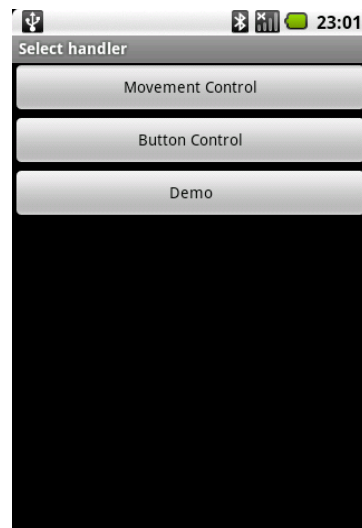


Figure 17:
HandlerSelectorActivity.

Following, each handler will be shown in detail.

1. Movement control



Figure 18: *MovementHandlerActivity*.

This activity is implemented by *MovementHandlerActivity* class. It allows the user to move the robot using the phone as a joystick. In this control mode, the phone should be held in landscape mode, in order to enhance control experience. When the phone is tilted forwards or backwards, the robot will start moving in the same direction. If the phone is tilted sideways, the robot will turn. When this activity is shown, a radar-like view is presented. This view shows an arrow which points the direction the phone is tilted, and its length grows or reduces depending on how far the phone is tilted.

Also, if the screen is tapped the recording mode will be started, recording all movements in a database.

If it's tapped again, it will stop recording and a dialog will be shown to the user in order to make him set a demo name. This demo can be reproduced later in *Demo* mode.

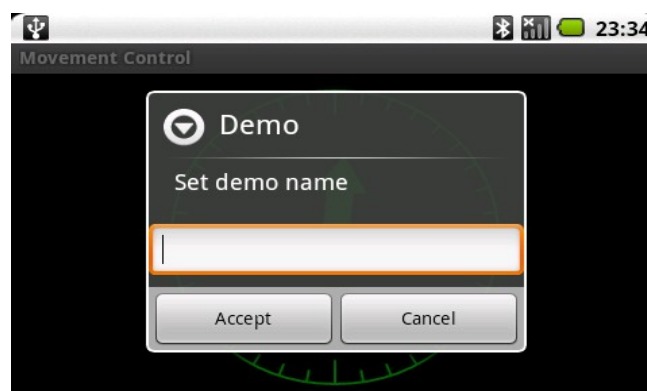








Figure 19: *MovementHandlerActivity*: save record.

2. Button control



Figure 20:
ButonHandlerActivity.

This activity is implemented by *ButonHandlerActivity* class. It allows the user to control the robot using a joy pad. Six buttons are presented in this activity. Each one has a image on it will indicates each button purpose

-  Makes the robot go forwards
-  Makes the robot go backwards
-  Makes the turn around left
-  Makes the turn around right
-  Stops the robot
-  Start/Stop recording mode

As seen in buttons descriptions, there is no possibility of making curves with the robot in this activity. When a button is tapped, the robot starts moving until the stop button is pressed.

Also, there is a seek bar view which is used to control the power applied to robot motors. When the seek bar is dragged right, the power increases, when is dragged left, the power decreases.

There is no need to stop robot movement to change the power, it can be changed while it is moving, and the effects will be seen immediately.

3. Demo

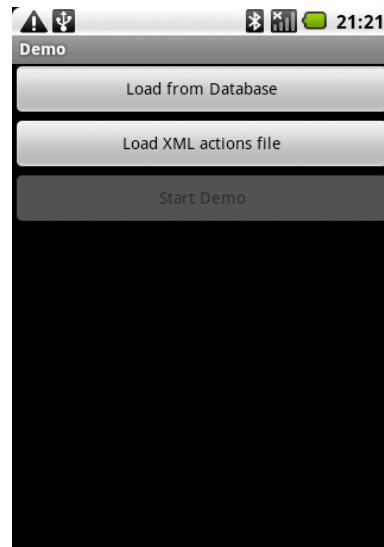


Figure 21:
DemoHandlerActivity.

Demo is the only activity which doesn't need user interaction to move the robot. It's implemented by *DemoHandlerActivity* and is also the only handler which doesn't use any controller since it has all robot control implemented.

In this activity, the user can load sets of movements from either previous movements recorded in *MovementHandlerActivity* or *ButtonHandlerActivity* or from an external XML files with a well-defined structure. These sets of movements are called **Demos** and can be reproduced in this mode.

When *DemoHandlerActivity* is launched, a window with three buttons is presented, as well as a black space in the bottom of the screen that will show the commands that are being sent to the robot (commands screen).

The first two buttons are used to load predefined movements from different sources (database or XML), while the third is only available when some demo is already loaded.

This activity allows data loading from different sources:

- **Database:** Allows the user to load a previously saved set of movements (Demo) from database. The button is labeled as *Load from database*
- **XML:** Sets of movements can be load from external XML files with a well-defined structure. The button is labeled as *Load from XML file*

Once a demo is loaded either from database or XML, the button labeled as *Start demonstration* is enabled.

Then, the activity starts sending all the movements to the robot until there are no more movements to send.

All the movements sent are shown in the commands screen while they are sent. Per each movement, the applied power, turn ratio, tacholimit and duration will be shown.

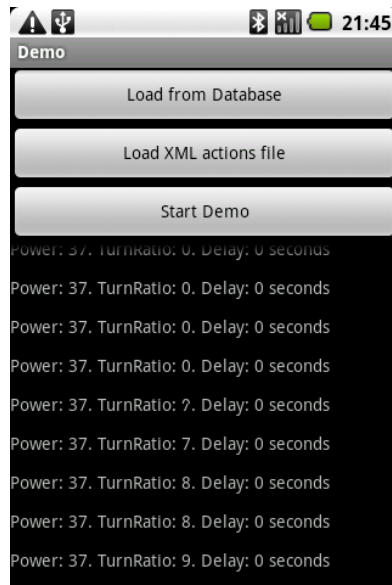


Figure 22:
DemoHandlerActivity:
running a demo

Following, both methods of demo loading will be shown in detail.

■ Database loading

When this button is pressed, the activity *DemoSelectorActivity* is launched. This activity reads all available demos from database and list them in a view.

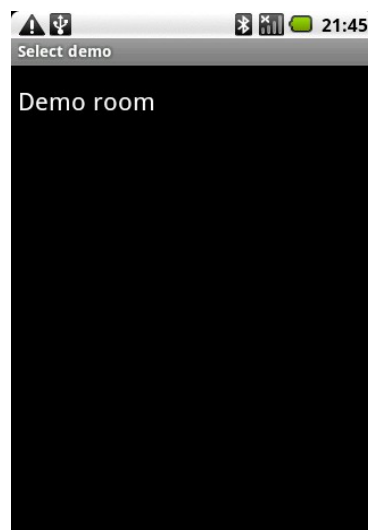


Figure 23:
DemoSelectorActivity: List
all demos available in
database

If the user performs a long click on a demo name, a contextual menu appears. This menu offers two options: **rename** (which changes the name associated to this demoID), and **delete** (which delete this demo and all associated actions). If the user presses the **Menu** button in the phone, a menu options appears offering the possibility of deleting all available demos. If the user touches a demo name, the demold associated to that name is passed back to the DemoHandlerActivity in order to load the demo.

DemoHandlerActivity receives the demold and enables *Start demonstration* button . If the user taps it, the activity starts a thread reads each action associated to the loaded demo, and sends a movement command to the robot with its parameters. Once there are no more actions available, the thread stops the robot and finishes execution.

■ XML Loading

When the XML loading is selected, the *FileExplorerActivity* is launched. This activity implements a file explorer that shows only XML files and directories. Root folder of this file explorer is the /sdcard folder which is the folder where the SD card is mounted. The user can navigate throughout the files of its sdcard and select the XML file of the demo he wants to load. The application then confirms the selection and passes the absolute path of the file to the *DemoHandlerActivity*.



Figure 24:
FileExplorerActivity: select
XML file

When the execution comes back to *DemoHandlerActivity*, it parses the XML file. As explained, the XML has to be well-defined. If the loaded XML doesn't follow this structure, it won't be loaded and a error will be shown to the user.

Demo XML Structure

```
<?XML version="1.0" encoding="UTF-8"?>
<demoactions>

  <!-- Each movement will be implemented in the following way.
        There should be a nxtaction tag which attributes will be
        the movement parameters that has to be sent to the robot
        These parameters are the following:

        - power: Power to be applied during this movement
        - turnratio: Turnratio to be applied during this movement
        - tacholimit: Travel distance in degrees
        - duration: duration of this movement in ms
  -->

  <nxtaction power="60" turnratio="0" tacholimit="0" duration="5000"/>
  <nxtaction power="60" turnratio="10" tacholimit="0" duration="6000"/>

</demoactions>
```

Each *nxtaction* tag has been implemented as a **DemoAction** object, which contains all the parameters of each entry. When the *DemoHandlerActivity* parses the XML file, it creates an *ArrayList* of *DemoAction* objects, one per each entry and will enable the *Start demonstration* button.

Once the user taps this button, the activity launches a thread that iterates over the *ArrayList* and sends movement commands to the robots based on each object parameters, updating the commands screen each time a new movement is sent. Once there are no more actions available, the thread stops the robot and finishes execution.

4.2.2 FOLLOWER MODE

The **follower mode** is an extension of the synchronized mode. It does not add any new handler nor controller, it adds the logic of controlling the leader behavior depending on the follower. This mode is available to all handlers since it is implemented in the *NXTHandlerBaseActivity*, and all handlers extends it.

If the phone is connected to two robots, follower mode can be enabled in the preferences activity. When follower mode is enabled, motor port where the IR emitter is connected must be selected. Now, if the user launches a handler, instead of launching the handler, the application launches the **FollowerConfiguratorActivity**. This activity lists all the connected devices allowing the user to select which robot will be the follower.

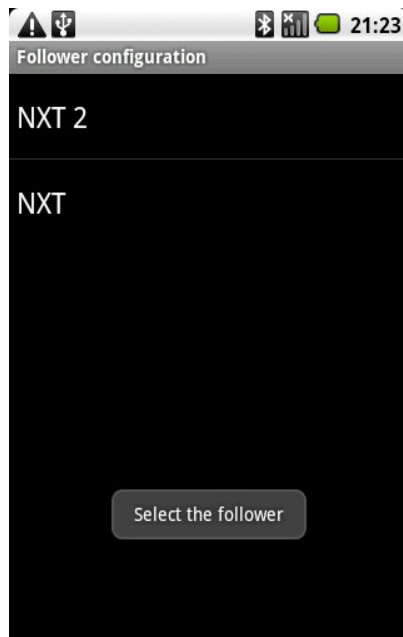


Figure 25:
FollowerConfiguratorActivity:
select follower.

Once the follower is selected, the application lists all the programs that are already installed in the robot. The user selects which program will be used for leader following and the activity starts it remotely. In order to list the programs installed in the robot, both *findFirst* and *findNext* methods of the *MiscInterface* object of the Lego FANTOM implementation are used. To start the program, *startProgram* method from *NXTMisc* is used.

Code to read files from the follower robot

```
response = fileReader.findFirst("*.rx");
fileNames.add(response.getFileName());
handle = (int)response.getHandle();
while(response.getStatus() != FindFileResponse.FILE_NOT_FOUND){
    response = fileReader.findNext(handle);
    nextHandle = (int)response.getHandle();
    fileNames.add(response.getFileName());
    fileReader.closeHandle(handle);
    handle = nextHandle;
}
fileReader.closeHandle(handle);
```

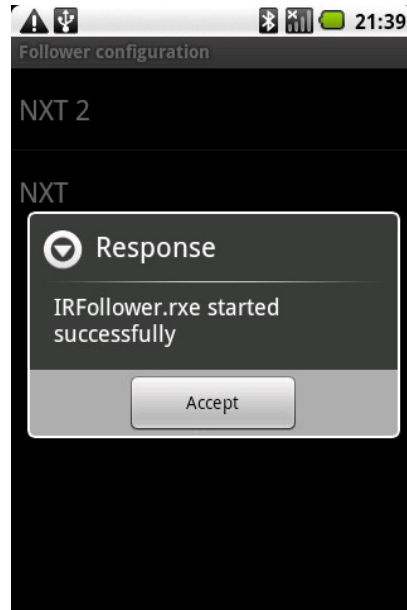


Figure 26:
FollowerConfiguratorActivity:
IRFollower program started.

Only when a program is started in the follower, the handler is launched. The handler detects follower mode and turns on the IR Emitter, and after, waits for follower initialization until a *READY* message is received. During this waiting, user interaction is disabled and user is notified by a phone vibration and a pop up message. When the activity receives this message, the handler enables user interaction and will continue as in synchronization mode. In order to enable and disable interaction, `disableHandler` and `enableHandler` methods of a handler are used.

If, during the leader movement, the follower gets lost, it sends messages to the phone, indicating that it has lost the leader and which was the last position that the leader was seen. In all these cases, user interaction is disabled and the user is notified.

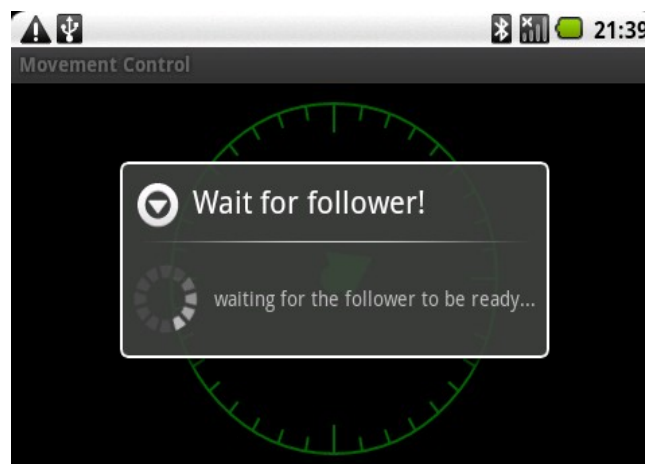


Figure 27: Follower mode: Wait for the follower to be ready.

In order to control when the follower gets lost, a simple protocol has been implemented. This messages are received by a thread. This thread listens for follower messages by an active waiting and when it receives one, it takes control of the execution. It disables user interaction, notifies the user, and moves the leader in the way explained in the above table until the follower finds it. For the application, the follower has not found the leader until a READY message has been received. Table 5 shows this messages in detail.

Message	Values	Meaning	Leader behavior
LEADER_LOST	Byte 0: 0x02 Byte 1: 0x00	The follower has lost the leader and notifies the phone and does a surveillance spin in order to find the leader	The leader is stopped.
LEADER_SEARCH	Byte 0: 0x02 Byte 1: Last known direction: - 0x04:Left - 0x05:Right - 0x04:Front	After the surveillance spin, the leader has not been found. The follower notifies to the phone, sending the last direction known direction of the leader	If the last known direction is front, the leader will go backwards slowly during 100 degrees. Otherwise, the leader will spin in the opposite direction as the one sent from the follower during 100 degrees
LEADER_FOUND	Byte 0: 0x04 Byte 1: 0x00	The follower has found the leader. It notifies the phone and moves towards it until reaches its back	Leader is stopped
READY	Byte 0: 0x00 Byte 1: 0x00	The follower is ready to continue.	Waiting messages are disabled and user interaction is enabled. The execution can now continue normally.

Table 7: Messages sent from the follower to the phone.

5 SMARTPHONE AND ROBOT INTERACTION

In this chapter, the interaction between the smartphone and the robot will be explained, detailing all the logic behind the translation from the user input to the robot movements.

The activities that control the robots somehow are called **Handlers**. All handlers extends from an abstract class called *NXTHandlerBaseActivity* that is the one which implements all the logic that all handlers must use. In this way, it is really easy to implement a new handler to control the robot. The only requirement in order to be able to use this logic is to implement this class.

A handler may use a controller, which is a class that implements the logic behind the robot control. All implemented controllers must extend *NXTBaseController* class. These controllers “translates” the user input (such phone movements, or button touch) into robot movements.

Following, these two base classes will be detailed in order to make clearer the explanation of the activities which control the robot from user inputs, such phone movements or buttons.

■ ***NXTHandlerBaseActivity***

All handlers must extend *NXTHandlerBaseActivity* abstract class, which implements all the logic related with basic robot control as well as the follower mode. It handles all connections and robot settings (such wheel ports or IR Emitter ports) as well as application modes.

When the application is used in synchronization mode, *NXTHandlerBaseActivity* maintains a unique connection object which is used to broadcast all messages to all connected robots. In case the application is used in follower mode, it handles two connection objects, one for the leader and one to the follower with the purpose of being able to send different commands to different robots.

All *NXTHandlerBaseActivity* subclasses must implement *enableHandler* and *disableHandler* methods. This methods will be called when there is a need to disable or enable user interaction. User interaction must be disabled, for example, when a new message is received from the follower, and must be enabled when the follower is ready to continue following the leader.

In turn, *NXTHandlerBaseActivity* extends the *Activity* class and overrides its main callback methods. The methods overridden by *NXTHandlerBaseActivity* are the following:

- **onCreate:** Called by Android system when the activity is created. In this callback method, *NXTHandlerBaseActivity* will initialize all basic objects (such as the preferences manager or the *BTManager*) and will check if this is the first time the application is started. In such a case, the preferences activity will be launched. Must be overridden by subclasses.
- **onResume:** Called by Android system when the activity resumes. In this method, the activity will check if any preference has been changed, and, in that case, it will set it in to the controller. Also, in case, follower mode is selected, will launch an the *FollowerSelectorActivity*, and activity used to configure this mode.
- **OnPause:** Called when the user is leaving the activity. In this method, the handler is disabled and after that, the robot is stopped as well as all pending recordings. Also, if follower mode is enabled, it turns the Ir Emitter off and stops the follower listener.

■ **NXTBaseController**

A Handler normally uses a controller class to interact with the robot. All controller classes must extend *NXTBaseController* in order to use the basic robot methods. *NXTBaseController* maintains all the robot configuration, such as the ports used for the motors, or the port used for the Ir Emitter (in case of follower mode). It also exposes methods to turn on and off the Ir Emitter.

The class automatically establishes wheels roles for motor synchronization, being the left wheel the master wheel and the right wheel the slave wheel. In this way, the developer doesn't have to take care of the parameters it has to use in order to move the robot. The developer just have to call the *move* method exposed by *NXTBaseController* passing both the turn ratio and the power.

This method will set received arguments as well as the rest of needed parameters to move the robot in the proper direction. Anyway, a full parametrizable move method is also exposed.

The class also controls movement recordings, the so called **demos**. Demo recording is implemented using the SQLite database which stores the movements. SQLite is a simple, lightweight and reliable transactional database engine that occupies a small amount of disk storage and memory, so it's a perfect choice for creating databases on many mobile operating systems.

All the movements are recorded in a simple database composed by two tables

- **DEMO:** contains the following columns: *demoID* and *demoName*. *demoID* is the primary key and is defined as an autoincrementable integer. *demoName* is the name given to the demo. By default this field value is "unnamed".
- **DEMO_ACTIONS:** contains the following columns:
 - *demoActionId*: Id of the action. Primary key and is defined as an autoincrementable integer.
 - *demoId*: Foreign key. Relates this action to one demo .
 - *power*: power to be applied to the robot during this action's duration.
 - *tacholimit*: travel distance in degrees to be applied to the robot during this action's duration.
 - *turnRatio*: turnRatio to be applied to the robot during this action's duration.
 - *duration*: duration in milliseconds of this action .

If the recording mode is enabled, *NXTBaseController* will save in the database all the movements sent to the robot. To measure the duration of each movement, a timer is started just before the command is sent. When the next movement is ready, the timer is stopped, and the movement is recorded with the duration calculated from the timer.

Following, all types of robot control will be explained along with the implementation of the handler and controllers that composes them.

5.1 CONTROL BY MOVEMENT

The application allows the user to control the robot by moving the phone. In this control mode, the user uses the phone as a joystick, and the application registers phone movements and translates them to robot commands.

Phone movements registered thanks to that all Android-Enabled phones are provided of sensors which can be read in order to know which way up the phone is, how far its tilted etc. The classes of readings available are the orientation (which way up the handset is, which way it is tilting etc), how fast it is moving and where it is. These are termed readings for ORIENTATION, the ACCELEROMETER and the MAGNETIC_FIELD.

Handlers can be written to read this sensors. The Handler has to register itself as a sensors listener and since then, it will be notified about this signals throughout its runtime. The Handler will be interrupted every time a new reading is available, so new sensor values can be used.

1. MovementHandlerActivity

MovementHandlerActivity is the handler which implements this kind of robot control. In order to be able to read sensors values, it implements *SensorEventListener* interface and its *onSensorChanged* method, which is the one called by the Android system when new sensor values are available. Each time it is called, it receives a *SensorEvent* object which contains the new sensor values [16].

MovementHandlerActivity also extends from *NXTBaseHandlerActivity* in order to be able to handle preferences change and be able to use the follower listener .

Three values are sent from the orientation sensors: Azimuth, Pitch and Roll.

- **Azimuth**, describes the angle between the magnetic north direction and the y-axis, around the z-axis (0 to 359). 0=North, 90=East, 180=South, 270=West
- **Pitch** describes the rotation along the x axis in degrees. This is measured from -180 to +180
- **Roll** describes the rotation along the y in degrees. This is measured from -90 to +90

By default the phone coordinate-system is defined relative to the screen of the phone in its portrait orientation. The axes are not swapped when the device's screen orientation changes.

The X axis is horizontal and points to the right, the Y axis is vertical and points up and the Z axis points towards the outside of the front face of the screen. In this system, coordinates behind the screen have negative Z values.

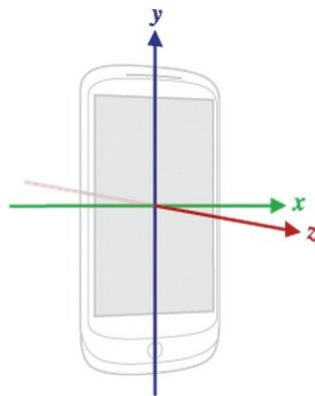


Figure 28: Android device coordinate system

In order to improve controls, the movement control is used in landscape mode, which implies that axes must be changed to the world's coordinate system.

In this mode, the power applied to robot motors is defined by the tilt along the Y axis, and the turn is defined along the tilt along the X axis.

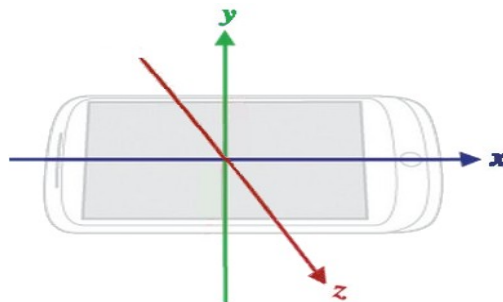


Figure 29: Android device coordinate system transformed to world coordinate system

MovementHandlerActivity transforms sensor values from the device coordinate system to the world's coordinate system in runtime. To make such a transformation, Android provides some methods useful for this purpose. Actually, ORIENTATION sensor values are calculated from ACCELEROMETER and MAGNETIC_FIELD sensor values, so instead of letting Android make the calculation using the phone coordinate-system, the activity does the calculation by itself.

Get and register ACCELEROMETER and MAGNETIC_FIELD sensors

```
// Get Android's sensor service
mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
// Get all sensors as a list
List<Sensor> listSensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);
// Get accelerometer sensor (first value of the sensors list)
mAcceleratorSensor = listSensors.get(0);
// Get magnetic sensor (second value of the sensors list)
mMagneticSensor = listSensors.get(1);
```

```
// Register both sensors
mSensorManager.registerListener(this,
    mAcceleratorSensor, SensorManager.SENSOR_DELAY_GAME);
mSensorManager.registerListener(this, mMagneticSensor, SensorManager.SENSOR_DELAY_GAME);
```

MovementHandlerActivity registers ACCELEROMETER and MAGNETIC_FIELD sensors instead of ORIENTATION sensor, and once it has values from both sensors, calculates the orientation values based on world's coordinate system and returns them as radians. The following code will return the new orientation coordinates based on the world's coordinate-system.

System coordinates translation

```
// Get rotation matrix to translate coordinates
SensorManager.getRotationMatrix(Rarray, Iarray, accels, mags);
/* Remap coordinates using the rotation matrix (first argument).
   It also receives the new X and Y axis.
   -In new coordinate system, X axis will be what in phone coordinates was Y axis
   -In new coordinate system, Y axis will be what in phone coordinates was X axis
   pointing the opposite direction
*/
SensorManager.remapCoordinateSystem(Rarray, SensorManager.AXIS_Y,
    SensorManager.AXIS_MINUS_X, outR);
// Get Orientation values based on the new coordinate system in mOrientationValues array
SensorManager.getOrientation(outR, mOrientationValues);
```

This new values must be filtered because they may contain noise from the sensors. On tested phones there were some jitters when the device was close to an electromagnetic field such those created by computers or televisions. This jitters produced weird readings, and therefore, weird reactions in the robot behavior

To filter this jitters, a *SensorAverageDamper* class have been created. This class adapts the sensor output values by damping them over a window of previous sensor values. The new sensor values will be the average over all sensor values in the window. The window size applied is 6. *SensorAverage damper* implementation can be seen in the source code.

Although these values are the rotation in degrees around and axis, actually they can be treated as the more convenient form to the application. One of the most popular way of treatment for this values is as rectangular coordinates, as if they define points on a graph relative to a fixed origin and axis. There values are then converted to polar coordinates to get the angle and the distance they describe. Both the angle and the distance will be used to determine the turn ratio and power applied to the robot motors.

The code for rectangular to polar coordinate conversion is the following

Rectangular to polar coordinate conversion

```
/* Convert from radians to degrees.
   mOrientationValues array contains the new orientation values:
   - mOrientationValues[0]: contains new Azimuth value (Z)
   - mOrientationValues[1]: contains new Pitch value (X)
   - mOrientationValues[2]: contains new Roll value (Y)
*/
for(int i =0; i < mOrientationValues.length; i++){
    OrientationValues[i] = (float) Math.toDegrees(mOrientationValues[i]);
}
```

```

// If there is no Y value, return
if (mOrientationValues[2] == 0) return;

// Angle = arctan(Y/X)
mAngle = (int) ( Math.toDegrees( Math.atan( mOrientationValues[2] / mOrientationValues[1] ) ));
// Distance = sqrt(Y^2 + X^2)
mTilt = (int) Math.sqrt((
    Math.pow(mOrientationValues[2], 2))
    + (Math.pow(mOrientationValues[1], 2)));

/* Tilt is magnified in order to adapt it to the phone movement (when the phone is
   almost vertical,
   it raises its maximum value)
*/
mTilt *= TILT_MAGNIFIER;
// Limit distance to 100, because is the maximum power value for the robot
if (mTilt > 100) mTilt = 100;

```

Both distance (mTilt variable) and angle (mAngle variable) are passed to the *NXTMovementController* to move the robot

2. NXTMovementController

NXTMovementController is a class which implements the logic behind the robot control based on phone movements. This class, as all controllers, extends from *NXTBaseController*, which enables it to use the basic movement functions.

When the *MovementHandlerActivity* has the sensor values ready, it passes them down to the *NXTMovementController* using the *moveNXT* method. This method receives both angle and distance and makes some checks to them to assure if the robot should be moved or not. It's noteworthy that these values are compared to the ones received from the previous movement. If new values are equal to the previous ones, no command will send to the robot. In this way, the number of transmissions are reduced in order to save phone and robot battery.

Before moving the robot, both distance and angle are checked to assure that they have a minimum value. Minimum values are established to avoid the control to be too sensitive to the user. Almost nobody can hold a phone perfectly flat on their hands, and if no minimum value is set, it will be very difficult to have the robot hold still.

Power value is the one which rules in robot movement. If distance value is in ranges from -11 to +11, no command will be send to the robot. Otherwise, angle value will be checked. If it ranges from -5 to +5, the command sent to the robot will have turn ratio value of 0, which means "move forward". If the angle is over this minimum limits, it will be used to set the turn ratio.

Range values received from *MovementHandlerActivity* ranges from 0 to 90, been 0 when the phone is completely flat, and 90 when it describes a 90 degrees angle with the X axis. As explained in LEGO Fantom section, the higher the turn ratio is, the less power is applied to the slave motor.

So, and taking into account the angle minimum value of 5, the turn ratio values applied from angle values will range from 5 to 90, almost the same range values the turn ratio has.

This means that, the more the phone is tilted to one direction, the higher the turn ratio is, causing the slave wheel to turn slower. And, due to the tribot configuration, the slower the slave wheel turns, the sharper the described curve by the movement is, causing the impression that the robot turns more as the tilt angle increases.

Turn direction will be defined by the angle sign. As explained in Lego Fantom protocol section, depending on the the turn ratio value sign, the turn ratio will be applied to one wheel or the other.

When the phone is tilted left, angle sign will be negative, and when tilted right, it will be positive.

NXTMovementController uses the default movement function from *NXTBase Controller*, which sets the left wheel as the master wheel. So, when the robot receives a negative turn ratio, wheels roles (master/slave) will be exchanged, making the left wheel go slower than the right wheel and therefore, making the robot turn left. When it receives positive turn ratio values, the right wheel (slave wheel) will go slower than the left wheel (master wheel), making the robot turn right. This implementation gives complete control feeling with phone movements.

Code to send movements to the robot

```
// Check if new values are different from the previous ones
if(power != mPrevPower || angle != mPrevAngle){
    // Check if the power value is greater than the minimum
    if(power > MIN_POWER || power < (MIN_POWER * -1)){
        if(angle > MIN_ANGLE || angle < (MIN_ANGLE * -1)) {
            // If angle value is greater than the minimum, use it to turn
            move(angle, power);
        } else {
            // If not, the robot should go straight ahead
            move(STRAIGHT_ANGLE, power);
        }
    }
}
```

5.2 CONTROL BY BUTTONS

Apart from controlling the robot using phone movement, the user can also use a control based in buttons. A joypad-like set of buttons is presented to the user. Each button has associated a basic movement such as move forwards, move backwards, turn around left, or turn around right. Each time the user taps one of these buttons, a basic order is sent to the robot with undefined duration. The command will be running until the user decides to stop it by pressing a stop button.

1. ButtonHandlerActivity

ButtonHandlerActivity which handles all button control. It creates all movement buttons and associates a *NXTButtonController* method to each one, depending on its purpose. Each time a button is taped, its action associated will be saved, and the corresponding

NXTButtonController method will be called. This activity extends from *NXTBaseActivity* in order to be able to handle preferences and to use the follower listener. It also creates recording button that, when tapped, starts the recording mode, recording all movements in a database. If it's tapped again, it will stop recording and a dialog will be shown to the user in order to make him set a demo name. This demo can be reproduced later in Demo mode.

In order to use the seek bar view, ButtonHandlerActivity implements SeekBar.OnSeekBarChangeListener interface and its onProgressChanged method. This method will be called by Android system each time the seek bar is dragged. When onProgressChanged method is called, the new progress value saved as the new power, and then, the saved action is checked in order to know which movement the robot is currently performing. Once the movement is known, its method is called again with the new power.

2. NXTButtonController

Like all controllers, *NXTButtonController* is a class which implements the logic behind the robot control and extends from *NXTBaseController* in order to use the basic movement functions.

This controller has five methods, one for each button in *ButtonHandlerActivity*. Depending on the purpose of this button, the move method from *NXTBaseController* is called with proper parameters in order to move the robot.

6 FOLLOWER IMPLEMENTATION: IR FOLLOWER

Because of the follower has to move autonomously, it has been programmed for that purpose. For programming the robot, the RobotC programming language has been selected.

RobotC [31] is a C based programming language with additional language extensions specifically for robotic use. It is the only one programming language for robots that includes a powerful Windows environment IDE with fully integrated software debugger that allows developers to step line by line through program execution and analysis of all variables. Additional debugging tools allow the user to see the real time states of all motors and sensor without the need of an external program. Also, includes an advanced source code editor with smart indenting, automatic code completion and a tabbed interface to allow multiple program to be open at the same time. In order to program MINDSTORMS robots with RobotC, a new firmware must be installed in the robot. This new firmware enhances Lego original firmware improving motor control and adding the debug possibility.

All remote control using LEGO MINDSTORMS NXT Communication protocol is working, but the use of motor synchronization, which is broken [33]. That's why, a robot using RobotC firmware can only be used as a follower. The application can connect to it, but no turns can be performed because of the lack of motor synchronization.

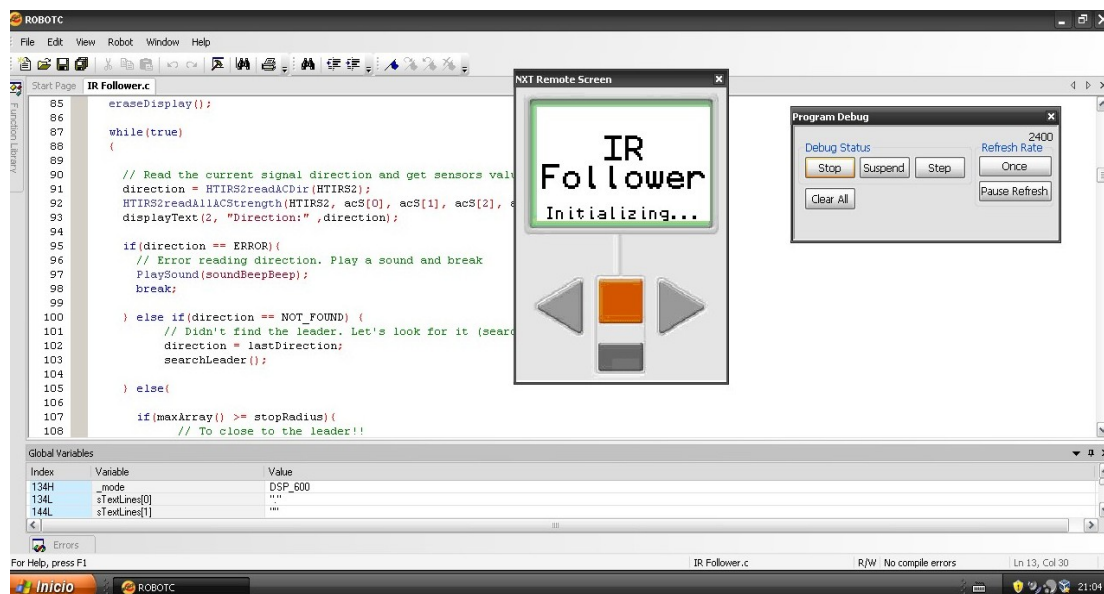


Figure 30: RobotC IDE with remote NXT screen enabled

6.1 ROBOT PROGRAMMING

The follower program uses the HITECHNICS IR Seeker V2 in order to track the leader's path. To use this seeker HITECHNICS provides drivers for RobotC with an API that provides several functions to read sensors values as well as the calculated direction.

As told in the IR Seeker V2 section, this sensor is divided in 9 zones that are not symmetrical, some zones are much narrow than others. The configuration chosen is to use one of the wide

zones as the center (zone 5) and two narrow for the sides. Therefore, three zones of action has been defined: center (zone 5), left (zone 4) and right (zone 6).

Zones 4 and 6 are calculated by the firmware by interpolating the values of two sensors. This two sensors are used to calculate the turn ratio that has to be applied to the wheels. The value difference is multiplied by a value, and then, the result is subtracted from the turn ratio value needed to make both wheels go synchronized. If the direction returned is different from this three zones (4, 5 or 6), it means that the follower has turned too much, so the follower turns around to that direction.

If returned direction is 0, the leader has been lost. It's important to remark that turn ratio definition in RobotC is different than in the Lego FANTOM protocol. In RobotC, a value of 100 for turn ratio means that both wheels should go to the same direction at the same speed. Also, the sign of the turn ratio doesn't affect wheels roles, only affect in slave wheel direction. If the sign is negative, the slave wheel will go backwards, if not, will go forwards.

Figure 31 shows IR Follower dataflow.

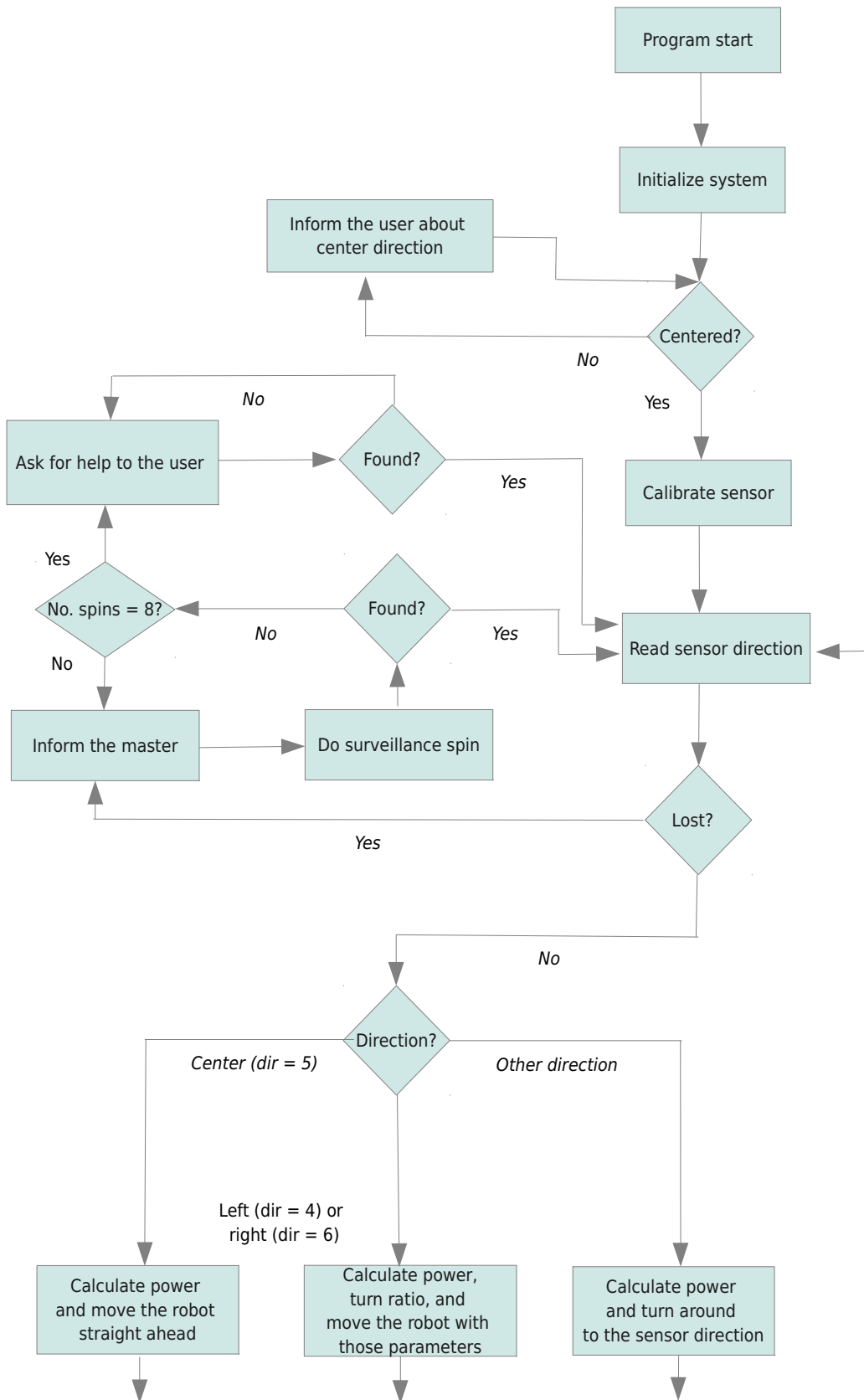


Figure 31: IR Follower dataflow

When the program is started, the user is requested to put the follower robot just behind the leader. It also gives instructions to help him to center the sensor to the IR Emitter. This instructions are shown in NXTs screen telling the user if the robot has to be moved left or right to be centered.

Once centered, the calibration starts. First of all, the sensor is set to register only 600hz signals. Also, sets wheels synchronization and configures wheels roles. Then, the follower starts turning around to each direction slowly, registering the maximum value for each direction. This value is saved in a array and it's used during movement to know if the follower is too close to the leader.

When the calibration has finished, the follower is ready to follow the leader. Then, it sends a READY message to the phone and waits for leader's movements. From now on, the program starts a control loop that reads continuously the direction and the sensor values from the seeker. Sensor values units indicate how strong is the signal they are receiving. The stronger the signal is, the higher the value is and therefore, the closer the leader is.

When the program has both direction and sensor values, it computes desired movement, always depending on the direction. To calculate the power that should be applied to the wheels, the program uses an algorithm based on the dynamic arrival from video games. For each sensor, a "slow down radius" is defined. This "slow down radius" is the sensor value that indicates that the robot should begin to slow down. During this radius, the desired speed is an interpolated intermediate value controlled by the sensor value. Outside this radius, the robot moves at maximum speed. The desired speed during the "slow down radius" is calculated as follows

Motor speed calculation

$$\text{speed} = \text{MAX_SPEED} * (\text{sensorSlowRadius}/\text{sensorValue})$$

where MAX_SPEED is 100 (maximum power for the NXT motors), sensorSlowRadius is the sensor value that indicates that the robot should begin to slow down, and sensorValue is current sensor value.

If the robot is outside the slow down radius, the division will be ≥ 1 (as the sensorValue will be lower than slowRadiusValue), so the robot will go at maximum speed.

TurnRatio algorithm is based on photovore algorithm for robotics. This simple algorithm is used for light follower robots.

When read direction is 4 or 6, both sensors used for this zones calculation are subtracted. Fuzzy logic has been used to calculate a multiplier value which will raise this difference up. The result of this operation is subtracted from the turn ratio value used to synchronize both wheels. The final result is the turn ratio that will be applied to the wheels during this zones.

Turn ratio calculation

```
// Get maximum value of both sensors
int maxS = max(acS[sensorPointer],acS[sensorPointer +1]);
// Get minimum value of both sensors
int minS = min(acS[sensorPointer],acS[sensorPointer +1]);
/* Multiply the difference by a sensor multiplier. Subtract the result to the turn ratio
value used to synchronize wheels */
int turnRatio = 100 - ((maxS - minS) * SENSOR_MULTIPLIER);
```

So, basically, the actions performed for each direction are the following

- **Zone 5 (center):** Leader is just ahead. Calculate power and apply to the wheels. The turnRatio applied will be 100.
- **Zone 4 (left) or zone 6(right):** The leader is turning. Calculate turn ratio and power and apply them to the wheels
- **Other zone:** Leader is too right or too left. Turn around in order to align to it
- **Zone 0:** Leader lost. Start searcher mode.

When the leader has been lost (zone 0 read), searcher routine is started. First of all, a LEADER_LOST message is sent to the phone, in order to inform the phone. Phone application then stops the leader and disables user input. From now on the algorithm varies depending on the last direction the leader was seen.

If direction is 5 (ahead), means that the follower went too slow to follow the leader. Then, the leader moves backwards slowly to be visible. When it becomes visible, follower sends a READY message to the phone indicating that he is ready to continue.

If direction is different than 5, the follower attempts to see the leader turning around 360 degrees. If leader wasn't seen, the follower sends a LEADER_LOST message, indicating the last known direction, and then the leader starts turning around in the opposite direction during 100 degrees of the robot wheels (around 45 degrees in flat).

For each leader's movement, the follower makes a 360 turn. This continues until the leader has reached a complete turn (the follower has made eight 360 turn).

Should this happen, the follower starts beeping to inform the user that the leader can't be seen by itself and needs user help. If the leader is found, a LEADER_FOUND message is sent to the robot in order to make it stop turning. Then, the follower moves to leader's back and sends a READY message. From this moment, the execution can continue normally.

7 TESTS AND RESULTS

In this chapter, several experiments are going to be described, in order to evaluate the developed application's functionality. In particular, robot control (synchronized mode) and follower mode are the main goals of the experiments. Also, recording and demo mode will be evaluated, as it is needed in order to test both functional modes. All tests have been made using full charged batteries in both leader and follower.

Following, **robot control** will be evaluated. For its evaluation, is important to confirm that **communications** with the robot are working properly, since one depends on the other. Therefore, both points will be evaluated at the same time.

For these evaluations, the application has been modified to poll motors state after each sent command. `NXTHandlerBaseActivity` has been modified in order to show on screen the response of the robot. In this way, both Bluetooth library and Lego FANTOM implementation are checked by testing if the parameters are correctly arriving to the robot. Robot control, which relies in communications, is also tested, since it generates the commands that have to be sent to the robot.

We are particularly interested in Power, TurnRatio and TachoLimit values, just because they are the only calculated values which change robot behavior

It's important to remark that this modification has only been implemented for testing. According to Lego FANTOM protocol documentation, robot response can introduce a latency around 60ms. This is highly noticeable in the application, because each time a command is sent, the application has to wait for the response, which introduces quite severe latency in application response. Lego firmware can not attend new messages while it is processing a response, so there is no possibility to have a different thread waiting for the response.

First of all, **control by movements** (`NXTMovementHandlerActivity`) is tested. We want to test that when the phone is tilted forwards and backwards, the power value changes. Also, the power value sent has to be positive when the phone is tilted forwards, and negative when the phone is tilted backwards. As explained in the Lego FANTOM protocol implementation, the power value depends on the rotation around the X axis, and the turn ratio value depends on the rotation around the Y axis.

In order to test the power applied to the robot, the phone is tilted forwards and backwards and both power values sent to the robot and the ones received from the robot are compared. The objective is to pass over the full range of power value, from -100, to 100.

After testing power values, turn ratio value is evaluated. The method followed is the same: compare sent values with the ones received from the robot. Since turn ratio values are independent from the power, it can be tested independently. The only rule is to have at least a minimum power, but this is not a robot constraint, this is implemented in the application to enhance robot control. Thus, the phone is tilted slightly forwards to achieve minimum required power, and then is tilted sideways. The same is done with the phone tilted slightly backwards, in order to test how the application behaves with negative power values.

Tables 8 and 9 shows both power and turn ratio values from the phone and the robot. The tables have been reduced because its full version would be too big for this document.

Phone power	Robot power
100	100
80	80
60	60
40	40
20	20
0	0
-20	-20
-40	-40
-60	-60
-80	-80
-100	-100

Table 8: Comparison of power values sent to the robot and received from the robot

Phone turnRatio	Robot turnRatio
90	90
70	70
50	50
30	30
10	10
0	0
-10	-10
-30	-30
-50	-50
-70	-70
-90	-90

Table 9: Comparison of turn ratio values sent to the robot and received from the robot. Power value of 60 is applied in all cases

As seen in Tables 8 and 9, the values received from the robot are the same as the values that the phone is sending. This means that the robot behavior is perfectly controlled by phone movements.

Following, **control by buttons** (ButtonHandlerActivity) is evaluated. Since we already confirmed that communications with the robot are working, testing objectives changes. We are interested of knowing if each buttons sends the correct order to the robot and if the seek bar changes the power. Table 10 shows buttons results.






Button	Behavior expected	Behavior observed
	Robot moves forwards	Robot moves forwards
	Robot moves backwards	Robot turns around right
	Robot turns around left	Robot turns around left
	Robot turns around right	Robot turns around right
	Robot stops	Robot stops

Tabla 10: Control by buttons: Behavior results

As seen in the table above, the behavior the buttons provoke in the robot are the ones expected.

Also, is important to check that the seek bar changes the power correctly. The robot will be moving forwards and backwards, and, meanwhile the seek bar will be dragged. The robot is expected to change its power while moving as the seek bar is dragged. If the robot is moving forwards, power values should be positive. If the robot is moving backwards, power values should be negative. Both power values sent and the ones received from the robot are compared in the following tables. Again, these tables has been reduced for this document.

Phone power	Robot power
100	100
80	80
60	60
40	40
20	20
0	0

Tabla 11: Control by buttons: Speed changes results moving forwards

Phone power	Robot power
0	0
-20	-20
-40	-40
-60	-60
-80	-80
-100	-100

Tabla 12: Control by buttons: Speed changes results moving backwards

We can conclude that user input robot control work properly since both act as expected. Lastly, tacholimit sending will be evaluated.

Due to handlers are implemented in such a way that don't use tacholimit value (since, is not possible to know the travel distance beforehand when the user is controlling the robot), it has been tested in DemoHandlerActivity using a XML file.

In this way, XML functionality of DemoHandlerActivity is tested as well by creating a XML file that contains a set of movements which makes the robot turn around some degrees.

As said in Lego FANTOM protocol implementation, this degrees are the degrees the wheel turns. In order to be sure that the robot doesn't stop turning because of the action duration has expired, this duration value has been set very high (around 10 seconds). The robot should stop turning around when it has traveled the specified degrees, although action duration is not expired.

Two points should be taken into account:

- The robot turns around but stops after some time.
- The tacholimit value sent is the same as the one received from the robot

Second point is important, because, we want to assure that the robot stops after the specified degrees regardless of action duration hasn't expired. Since tacholimit value is a long value, all its possible values can't be tested, since its range is very high. Therefore, only a subset of possible values has been tested.

Table 13 shows the results for tacholimit

Phone turnRatio	Robot turnRatio
100	100
800	800
1500	1500
5000	5000
10000	10000

Table 13: Comparison of tacholimit values sent to the robot and received from the robot

Table 13 demonstrates that tacholimit is correctly sent to the robot. Also, XML files are well read and processed.

After testing synchronized mode, the **recording mode** is evaluated. The result of this experiment will be useful in order to test Follower mode later. For testing recording mode, three paths have been designed.

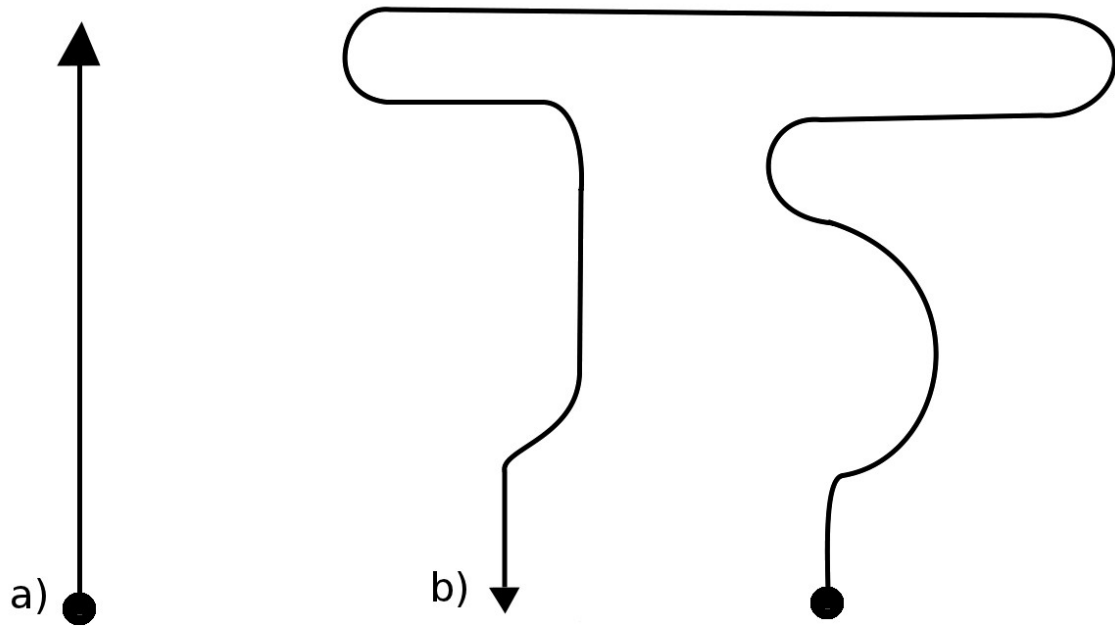


Figure 33: Follower mode paths: a) Straight line to test speed changes. b) Combination of serpentine curves with straight lines

The first one (a) is just a straight line where the leader changes its speed linearly. It starts from 0 power and increases it to 100. Then it decreases again to 0. This behavior is repeated 10 times during 11 seconds. The follower is expected to accelerate when the leader is going faster and decelerate when it is slowing down.

The second path (b) combines serpentine curves of different degrees with some straight lines. The objective with this second circuit is to see how it behaves with curves and if it gets lost. In this last case, "Leader lost" protocol should be activated.

This second circuit has been made at fixed power of 60 with full battery in both leader and follower. The duration of the complete path is around 54 seconds.

Both circuits are repeated five times. Tables 14 and 15 show the results of these experiments

Attempt	No. times the follower gets lost	No. of times the follower needs user help
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0

Table 14: Follower mode results: Straight line path with accelerations and decelerations

Attempt	No. times the follower gets lost	No. of times the follower needs user help
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0

Table 15: Follower mode results: Serpentine curves at fixed power of 60

As seen in the results, the follower does not get lost in proposed paths, so the "Leader lost" protocol couldn't be tested. In order to test it, a new more complicated path is designed.

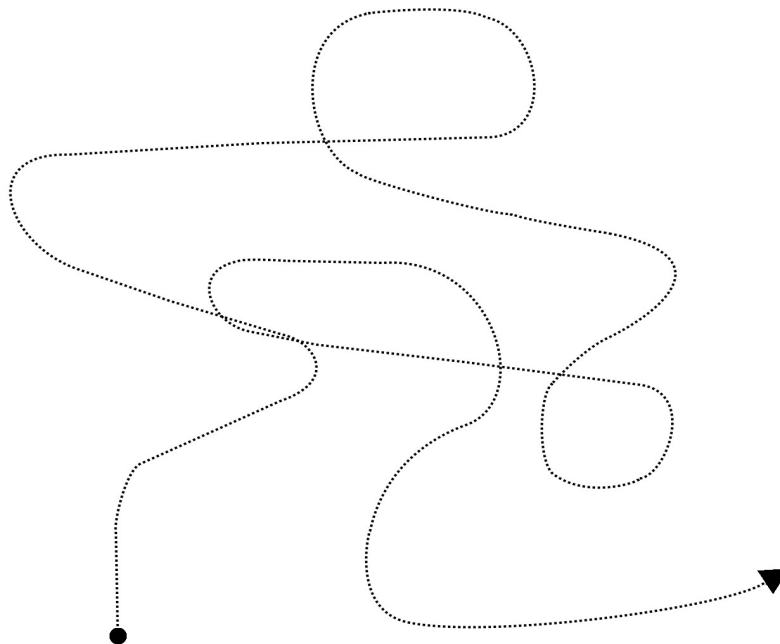


Figure 34: Follower mode: Path to test "Leader lost" protocol

Unlike the other paths, this one has been made at maximum speed (100). The complete path has a duration of 54 seconds. Table 12 shows the results of follower mode for this path

Attempt	No. times the follower gets lost	No. of times the follower needs user help
1	3	1
2	2	1
3	2	1
4	3	1
5	0	0

Table 16: Follower mode results: Path to test "Leader lost" protocol

In table 16 we can see that the follower needs such a complicated path in order to get lost. The maximum times it got lost is three, and almost in all the circuits, it needed user help once. The majority of losses are due to the leader goes much faster than the follower. However, as the results show, "Leader lost" protocol can recover the lost most of the times.

The only time when the follower needs user help is when the leader performs loops at speed higher than the follower, because there is a moment when the leader stays perpendicular to the follower. In that moment, the IR Emitter is not visible by the follower due to the leader's shape, which hides it. The last direction registered by the follower is ahead, since the leader has not changed its direction angle, but its rotation angle, and, for the IR sensor, the IR emitter stays in the same position. In such a situation, "Leader lost" protocol tries lost recovery from ahead direction, making the leader to go backwards, when it should turn around. This problem should be addressed easily by changing leader's build configuration.

As seen in results, the application can control the robot successfully. Both control methods have been tested correctly, which proves that both Bluetooth library and Lego MINDSTORMS NXT Communication protocol implementation are working properly. Control by movement has been tested by checking that power and turn ratio values sent to the robot depends on specific movements made with the phone.

Control by buttons works correctly as it sends basic movements when the proper buttons is pressed. In both cases, all values sent to the robot are arriving as they should, and therefore, we can conclude that robot behavior is totally controlled by the phone.

Regarding to Follower mode and "Leader lost" protocol, we can conclude that both work as expected. The follower is capable of following leader's path in almost all situations. The experiments show that the leader get lost very few times and in extreme situations, such those which imply very complicated paths at top speed. In case of lost, "Leader lost" protocol is capable of recovering in the majority of cases, needed user help only in specific situations such as loops. However, this problems can be addressed easily by changing robot build configuration.

8 CONCLUSION

In this document, we demonstrate that Bluetooth-based mobile systems might be a candidate in order to control autonomous systems wirelessly. DroidStorm was presented, as an experimental Bluetooth-based application developed for Android-Based mobile phones which provides different human interfaces to control Bluetooth capable autonomous systems such as Lego MINDSTORMS NXT robots. Furthermore, this application shows that mobile systems can create and control collaborative robotic system where the robots collaborate together to reach the same objective.

The system is composed by a Android-based smartphone which controls the Lego MINDSTORMS NXT robots by means of Bluetooth wireless technology. We developed a new Bluetooth library from scratch, by using BlueZ libraries embedded in the operating system, as Android didn't have a Bluetooth API until version 2. The two layers that compose this library were presented: the native layer coded in C language, which interacts with the BlueZ library, and the Java layer, which is used by the Android application to interact with the native layer. Both native and Java layer communicates by means of the JNI framework. This library is compatible with all Android versions.

Also, we wrote a Lego MINDSTORMS NXT Communication protocol library for Android in order to be able to control the robot remotely. This protocol uses Bluetooth Serial Port Profile (SPP) to communicate the robot with outside devices such as mobile phones or computers.

Application operating modes were presented: synchronized mode and follower mode. In synchronized mode, the movements of up to seven Lego MINDSTORMS robots can be controlled synchronously by sending all movement commands to all connected robots. This mode includes all available human interfaces with the robot. Control by movement mode allows the user to control the robot using the phone as a joystick by means of movement sensors included in Android phones. This control mode has to be used in landscape mode. The application gathers all the values returned by these sensors and calculates the movement command that will be sent to the robot. Depending on where the phone was tilted and how far, the robot will move to different direction at different speed. Control by buttons mode allows the user to control the robot using a joypad-like set of buttons. This buttons send basic behavior movements (such as move forwards, move backwards or turn around) and allows speed control. In any of these interfaces, the user can record the movement commands he is sending to the robot in order to be able to reproduce them whenever he wants. This movements are stored in SQLite database we designed for the purpose. The user can also load sets of movements from XML files with a well-defined structure.

Follower mode establishes a collaborative robotic system when two robots are connected. In this mode, the phone controls only one robot and the other one follows it autonomously. The phone establishes a leader/follower relation between the robots where the robot controlled by the phone becomes the leader and the other becomes the follower.

For the implementation, we built a IR emitter which emits modulated IR signals at 600Hz. This emitter is the point of reference for the follower and therefore has been installed in the leader's back.

The follower uses a IR sensor capable of filtering modulated IR signals in order to avoid external sources of noise (such as the sunlight). We programmed follower robot in RobotC programming language to follow leader's path based on the sensor values.

In order to control when the follower has lost the leader, we developed . This protocol a new protocol, called "Leader lost protocol". This protocol recovers the follower from leader loses autonomously. The follower uses this protocol to notify the phone that it has lost the leader

and where was the last direction it was seen. Depending on the type of loose and the last direction, user interaction is disabled and the phone moves the leader to help the follower to find it. This protocol covers all possible states of looses, since the moment when the leader has been lost, until it has been found.

Both robot control and leader's tracking where evaluated. First, communications between the phone and the robots where tested, in order to assure that both Bluetooth and Lego MINDSTORMS NXT Communication protocol library were working properly. We were interested in power, turn ratio and distance to travel values, which are the main variables which change robot behavior For this, motors state were pulled after sending each command, and then they were compared. Commands were sent using both user interfaces (movement and buttons), as user interface relies on communications. This gave us the opportunity to test both things at the same time. Results show that all messages are arriving properly and that the interfaces are sending the commands as expected.

After communications and robot control, recording was evaluated. For this, different paths where recorded. Since later this movements can be reproduced, we were able to test if movements were recorded correctly. Experiments showed that recording is working properly, as all proposed paths were reproduced successfully.

Finally, follower mode performance was evaluated. We were particularly interested in knowing how the built IR emitter performs as well as if the follower is capable of following the leader. For this, different predefined paths were used to test how many times the follower got lost and how many times it needed user help. First of all, two simple circuits where reproduced five times each. These circuits combined straight lines and curves. In the first circuit, the leader performs accelerations and decelerations in straight line in order to test how the follower behaves in such a situation. The second circuit introduces curves to test follower turns and to try to make it lost. This second circuit was performed at fixed speed of 60 in order to simplify tests. Results show that the follower can follow the leader perfectly in such a situations since it didn't get lost any time. After, a new more complicated path was used. This one introduces loops as well as increases leader speed. This path was performed at maximum speed (value of 100). Three was the maximum number of times the follower got lost, and it only needed user help once. As seen in results, the "Leader lost" protocol can recover from the majority of looses, needing user help in specific situations like when the leader stays perpendicular to the follower. However, such problem is due to leader's build configuration, since it's shape hides the IR emitter in that position.

Further research work should be done to improve "Leader lost" protocol in order to be able to recover from more difficult and different situations . Also, some research could be done with the IR emitter to see if it's performance can be improved and, in such a case, how it affects to the overall system.

9 REFERENCES

1. T. Kubik, M. Sugisaka, "Use of a cellular phone in mobile robot voice control", Oita University of Japan, Wroclaw University of Technology, 2001
2. Yu Chan Cho, Jae Wook Jeon, "Remote Robot Control System based on DTMF of Mobile Phone", SungKyunKwan University, Suwon, Korea
3. Ali Sekmen, Ahmet Bugra Kosku, Saleb Zein-Sabatto, "Human Robot Interaction via Cellular Phones", Tennessee State University (USA), METU Ankara (Turkey), 2003
4. O. Rogalla, M. Ehrenmann, R. Zollner, R. Becher and R. Dillmann, "Using Gesture and Speech Control for Commanding a Robot Assistant", Universitat Karlsruhe (TH) (Germany), 2002
5. iRobot Roomba. Webpage: <http://www.irobot.com/>
6. Sony Aibo. Webpage: <http://support.sony-europe.com/aibo/>
7. Lego Mindstorms NXT. Webpage: <http://mindstorms.lego.com/en-us/Default.aspx>
8. HTC Dream. Webpage: <http://www.htc.com/es/product/dream/overview.html>
9. Vishay TSUS5400 Infrared Emmitter. Datasheet available at: <http://www.vishay.com/docs/81056/tsus5400.pdf>
10. Hitechnics IR Seeker V2. Webpage: <https://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=NSK1042>
11. First Tech Challenge. IR Seeker V2 Characteristics. Document available at http://usfirst.org/uploadedFiles/Community/FTC/Team_Resources/IRSeekerV2characteristicsR2-rev3.pdf
12. Bluetooth Serial Port Profile overview - Palowireless. Available at: http://www.palowireless.com/infotooth/tutorial/k5_spp.asp
13. RobotMag. Webpage: <http://find.botmag.com/100701>
14. XDA-Developers: Webpage: <http://www.xda-developers.com>
15. Led Flasher circuit - Electronic circuits for beginners. Available at: <http://electroniccircuitsforbeginners.blogspot.com/2009/04/led-flasher-circuit.html>
16. 555 as astable - Electronics club. Available at: <http://www.kpsec.freeuk.com/555timer.htm#astable>
17. 555 calculator. Available at: <http://freespace.virgin.net/matt.waite/resource/handy/pinouts/555/>
18. Android activity lifecycle. Webpage: <http://developer.android.com/reference/android/app/Activity.html>
19. Android Sensor Event. <http://developer.android.com/reference/android/hardware/SensorEvent.html>
20. Android developer guide. Webpage: <http://developer.android.com/guide/topics/fundamentals.html>

21. Sensor arrow tutorial - Android academy. Available at:
<http://www.androidacademy.com/1-tutorials/43-hands-on/114-sensor-arrow>
22. Android developers group. Webpage:
<http://groups.google.com/group/android-developers?pli=1>
23. Android Native Development Kit (NDK) developers group. Webpage:
<http://groups.google.com/group/android-ndk>
24. Android developers group. Only Droid Support Bluetooth API?. Available:
http://groups.google.com/group/android-developers/browse_thread/thread/f8fa2b873e6d7e27#
25. How to enable/disable bluetooth programmatically - Anddev forums. Available at:
<http://www.anddev.org/viewtopic.php?p=14671>
26. Android 1.6 source code. Available at: <http://www.netmite.com/android/mydroid/1.6/>
27. Android-Spa forums. Webpage: <http://www.android-spa.com/index.php>
28. Android Bluetooth hacking using the NDK. Available at:
<http://pythonstring.myloger.com/2009/07/26/stephans-hacks-musings-android-bluetooth-hacking-using-the-ndk/>
29. Albert Huang. "An Introduction to Bluetooth Programming". Available at:
<http://people.csail.mit.edu/albert/bluez-intro/>
30. Lego Mindstorms NXT Bluetooth library in C for Linux Webpage:
<http://www.quietearth.us/nxtlibc.htm>
31. Android Sensors study - Mystic Lake software blog
<http://blog.mysticlakesoftware.com/2009/07/sensor-accelerometer-magnetics.html>
32. Having Bluetooth issues?. HTC Forums. Available at: <http://community.htc.com/na/htc-forums/android/f/91/p/2336/8570.aspx>
33. Rick Rogers, John Lombardo, Zigurd Mednieks, Blake Meike "Android Application Development: Programming with the Google SDK", O'Really, 2009
34. RobotC. Webpage: <http://www.robotc.net>
35. Direct commands and RobotC 2.0.. is motor sync possible?. RobotC forums. Available at:
<http://www.robotc.net/forums/viewtopic.php?f=1&t=3214&p=13531&hilit=direct+command+sync#p13531>
36. RobotC drivers documentation. Webpage:
<http://rdpartyrobotcdr.sourceforge.net/documentation/index.html>
37. Lego MINDSTORMS NXT Bluetooth developer kit. Available at:
<http://mindstorms.lego.com/en-us/support/files/default.aspx>
38. Michael L. Gasperi and Philippe "Philo" Hurbain, "Extreme NXT", Technology in action, 2009
39. Albert W. Schueller. Walla, Washington "Programming with Robots", Department of Mathematics, Whitman College in Walla, 2010