POLITECNICO
MILANO 1863

Embedded Systems
Advanced Operating Systems Project

Whistle Robot

Version 0.0
??/?/2018

Borgo Daniele	894110
Fusca Yuri	898602
Indirli Fabrizio	899892

Contents

1 Introduction

1.1 Purpose

The goal of this project is to design a game for 2 players, where the STM32s microphone is used to control a robot, and the player who can defeat the opponent robot wins.
This project is a variant of the project proposal where the STM32F4 was used in a game for 2 or more players to choose and recognise a frequency with the microphone.
In our project, we use the embedded microphone of the board to controls the stepper motors and move the robot on the surrounding environment.

1.2 Description of the robot

The robot will have a cylindrical shape and it will be equipped with wheels on the bottom that will allow it to move forward, backward and rotate on itself.
The engines that will allow the wheels to rotate are directly controlled by the STM32F4, the brain of our robot. We will use 3D printer to build the body of the robot.

1.3 Game logic

The game consists of a competition between 2 or more players. The players will control their robot through sounds, with the aim of attacking and overturning the opposing robot.
A specific movement of the robot will correspond to each sound (frequency):

- Forward movement
- Backward movement
- Rotation on itself

The rotation on itself is useful to turn around and attack the opposing robot

1.4 Code structure

We decided to structure the code in many components, in order to make the project extendable, easy to edit and easy to readapt.
The main core of the robot is represented by the controller. All other components of the robot will be connected to the controller.
The frequency analyser will convert the analog data obtained with the microphone into digital numbers.
The controller will receive the information from the frequency analyser and it will use this information to directly control the wheels of the robot.

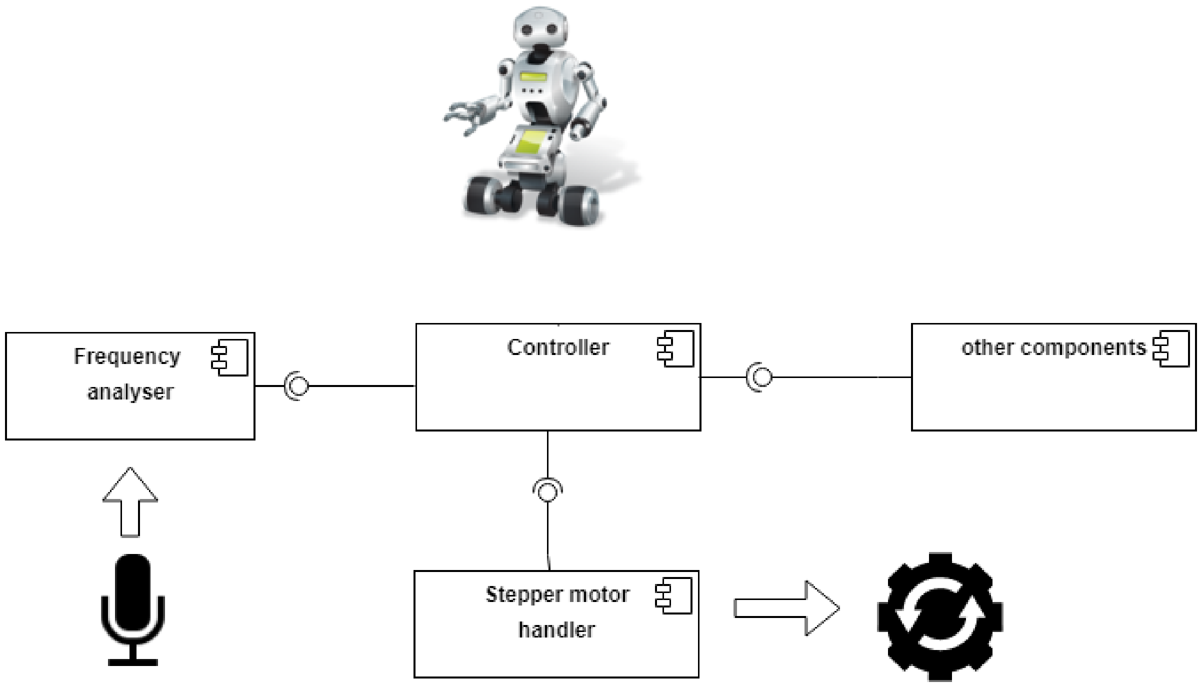


Figure 1: This is a general overview of the system.

2 Transmitter

The transmitter is the part that provides to convert the received sounds signals in commands for the robot, through a microphone. The STM32D407VG-Discovery board is equipped with a MP45DT02 MEMS microphone and a CS43L22 DAC for audio acquisition and processing. In this project, these devices are used to detect the sounds' frequency, which will determine the commands given to the robot.

2.1 The MP45DT02 microphone

The MP45DT02-M is a compact, low-power, omnidirectional, digital MEMS (Micro Electro-Mechanical Systems) microphone.

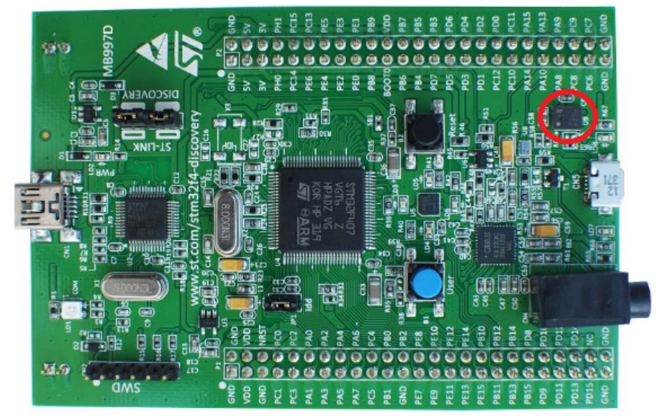


Figure 2: View of the upper surface of the board, with the microphone highlighted in the circle.

In the microphone bottom surface, there are six pins:

1. **GND**: Connected to the GND of the board.
2. **LR** - Channel selection: used since the microphone is designed to allow stereo audio capture. If it is connected to GND, the MP45DT02 is placed in *left* channel mode: a sample is latched to the data output pin (PDM, Pulse-Density Modulation) on a falling edge of the clock, while on the rising clock edge, the output is set to high impedance. If the LR pin is connected to Vdd then the device operates in *right* channel mode and the MP45DT02 latches it's sample to PDM on a clock rising edge, setting the pin to high impedance on the falling edge of the clock.
3. **GND**: Connected to the GND of the board.
4. **CLK** - Synchronization input clock: this pin is connected to the PB10 port of the board. The clock signal determines the sampling frequency.

5. **DOUT** - PDM Data Output. Connected to the PC03 pin of the board's GPIO.

6. **VDD**: Power supply.

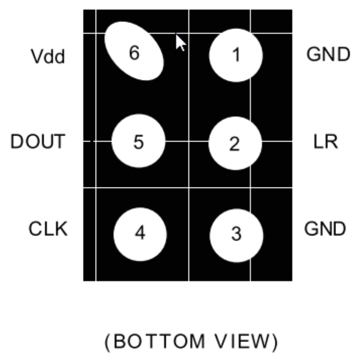


Figure 3: Pins on the bottom of the microphone.

2.2 Overview on sound acquisition and processing

The sound is acquired by the microphone, and the communication is performed through I2C.
The microphone acquires data in PDM format: each sample is a single bit, so the acquisition is a stream of bits; an high amplitude is represented with an high density of *1* bits in the bitstream.
The sampling frequency of the ADC is 44000 Hz.
In order to process the data, it has to be converted in PCM (Pulse-Code Modulation) format: this is done by doing *CIC filtering* (Cascaded Integrator-Comb), with a decimation factor of 16 (each 16-bits sequence of 1-bit PDM samples is converted in one 16-bit PCM sample).
Then, an FFT (Fast Fourier Transform) analysis is performed on 4096 samples at a time to extract the fundamental frequency and the amplitude of the sound.

2.2.1 Microphone initialization

The initialization of the sound acquisition system comprehends several steps:

1. SPI, DMA and General Purpose ports B and C are enabled through RCC

2. GPIO ports are configured in *Alternate* mode

3. SPI is set to work in I2S mode

4. interrupt handling for DMA is configured

2.2.2 Sound acquisition and processing

Each time a new block of samples is read, the *freq_recognition* library:

1. Reads blocks of sixteen bits from SPI and transfers them in RAM through DMA;
2. Converts 16 PDM samples in one 16-bit PCM sample via *CIC filtering* with a decimation factor of sixteen;
3. Perform FFT through the *arm_cfft_radix4_f32* module;
4. Calculate amplitude of each frequency with the *arm_cmplx_mag_f32* module;
5. Calculate the the fundamental frequency (and its amplitude) in the vector;
6. Store the values of the fundamental frequency and its amplitude in dedicated variables, then call the *callback function* to react to the detected frequency.

In this project, the *callback function* is in charge of producing commands to the engines when the frequency of the last acquired sample is in some specified ranges. These commands are sent to an appropiate object, named *ReceiverState*, which provides to sent the commands to the Arduino board. This object avoid the transmission of overwhelming sequences of the same string, sending only the first of an all equal string list.

2.3 The *freq_recognition* library

This library defines a simple interface for recording audio with the embedded microphone on the STM32F4 Discovery board and for performing an FFT analysis to calculate the strongest frequency (and its amplitude) of each sample.

2.3.1 The Microphone.cpp driver

Sound acquisition from the embedded microphone is performed by the ***Microphone.cpp*** driver.
This driver was originally developed by Riccardo Binetti, Guido Gerosa and Alessandro Mariani, but it has been improved by Lorenzo Binosi and Matheus Fim in their *Digital Guitar Tuner* project.
For this project, some little changes were made to the driver, such as reducing the decimation factor to improve sampling frequency.

2.3.2 Interface of the library

The *freq_recognition* library provides the following functions:

- ***void startAcquisition(function<void ()>cback, float32_t* freqVar, float32_t* amplitudeVar)***, whose parameters are:

  - ***cback***: a pointer to the callback function that will be executed automatically each time a new sample has been acquired and processed;
  - ***freqVar***: pointer to the float32_t where the detected frequency of each sample will be stored;
  - ***amplitudeVar***: pointer to the float32_t where the detected amplitude of each sample will be stored.

- ***void stopAcquisition()***: stops the acquisition of new samples.

2.3.3 How it works

The *freq_recognition* library wraps the microphone driver and adds the FFT analysis features.
When the *startAcquisition()* method is called, the library retrieves the Microphone object (which is a singleton) and configures it to execute the *FFTandCallback* function each time a new chunk of PDM samples is produced (each chunk has FFT_SIZE samples). Then, the *FFTandCallback* function:

1. Initializes the ARM Complex-FFT module;
2. Calculates the FFT, producing a vector (called *input[]*) of complex numbers;
3. Calculates the magnitude of each number in the *input[]* vector and stores the values in the *output[]* vector;
4. Calls the *maxFreq()* function to calculate the strongest frequency: this is done by finding the position of the greatest value in the *output[]* vector, which is then multiplied for the frequency resolution, which is SAMPLING_FREQ / FFT_SIZE;
5. Calls the callback function specified as parameter of the *startAcquisition()*.

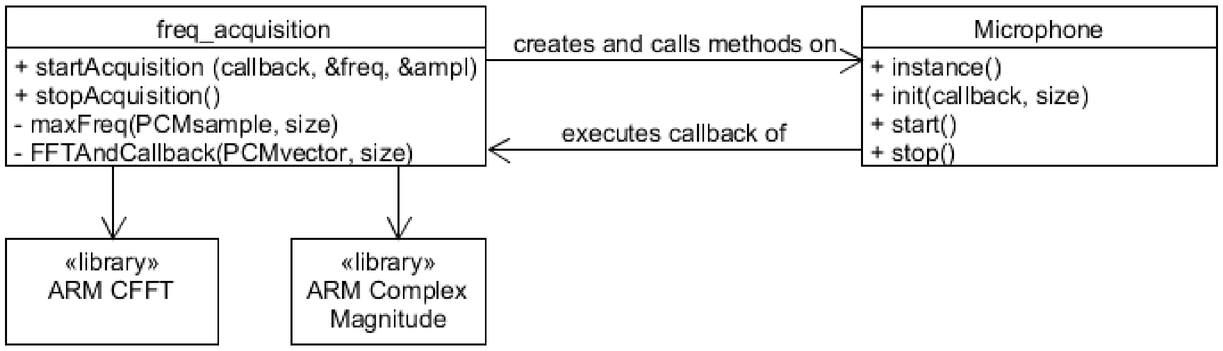

freq_acquisition
+ startAcquisition (callback, &freq, &ampl)
+ stopAcquisition()
- maxFreq(PCMsample, size)
- FFTAndCallback(PCMvector, size)
creates and calls methods on
executes callback of
Microphone
+ instance()
+ init(callback, size)
+ start()
+ stop()
«library» ARM CFFT
«library» ARM Complex Magnitude

Figure 4: UML schema of the interaction between the modules in frequency recognition.

2.3.4 Usage of the library

The usage of the library is performed according to these steps:

1. Add the ***freq_recognition.cpp*** and ***freq_recognition.h*** files in the folder of the other source files of the project, and add the *freq_recognition.cpp* entry in the **SRC** line of the makefile;

2. Add the *arm_bitereversal.c*, *arm_cfft_radix4_f32.c*, *arm_cfft_radix4_init_f32.c* and *arm_cmplx_mag_f32.c* libraries to the project (these libraries are required by *freq_recognition.cpp*);

3. Add the *#include "freq_recognition.h"* line at the start of the source file that will use the library;

4. Declare 2 float32_t variables to store the frequency and the amplitude of the last acquired sample;

5. Define a void callback function without parameters;

6. Start the acquisition by calling the *startAcquisition(&callback, &freq, &amplitude)* method, passing to it the addresses of the callback function and of the 2 variables. This method is non-blocking (operations are performed in other threads).

Each time a new block of 16-bit PDM samples has been acquired, the driver will perform an FFT analysis on it and will write the frequency and the amplitude of the sample in the chosen variables, then the callback function will be called automatically. This process will be iterated continuously until the *stopAcquisition()* function is called.

2.4 Displaying the recognized frequency

The transmitter is equipped with a standard HD44780 LCD with two rows and sixteen columns, that is used to print the currently recognized frequency and the associated command (if any).

2.4.1 Connections

LCD pin	**Connected to**
(1) Vss	GND
(2) Vdd	5V
(3) V0	center tap of a potentiometer between 5V and GND
(4) RS	PE7 pin of the Discovery board
(5) RW	GND
(6) E	PE8 pin of the Discovery board
(7) D0	5V
(8) D1	5V
(9) D2	5V
(10) D3	5V
(11) D4	PE11 pin of the Discovery board
(12) D5	PE12 pin of the Discovery board
(13) D6	PE13 pin of the Discovery board
(14) D7	PE14 pin of the Discovery board
(15) A	3.3V
(16) K	GND

2.4.2 Driver and usage

The driver of the display for Miosix is available in the OS itself: to use it, the *utils/lcd44780.h* library has been included in the main program.
In the software, the display is represented as an instance of the *Lcd44780* class, whose constructor needs to know the pins of the board on which the RS, E, D4, D5, D6, D7 display pins are connected. Through this object, these methods are available:

- *clear()* to wipe the content of the screen;
- *go(column, row)* moves the cursor to the specified position;
- *printf()* prints to the display, using the standard C formatting rules.

To reduce the risk of printing errors and the required number of updates, the *Lcd44780* instance is encapsulated in an class, named *Display*, in the homonym package. This class exposes only two methods: the first is to set the frequency and the others to set the commands. When one of these are invoked, it will update the display if and only if the passed value is not yet shown.

3 Wireless Communication

To bring the signals from the transmitter to the receiver, it is necessary using a wireless connection, in this case implemented trough two HC-05 modules. These boards are, in tecnical terms, Bluetooth SPP (Serial Port Protocol) modules, used to abstract the wireless connection implementation.

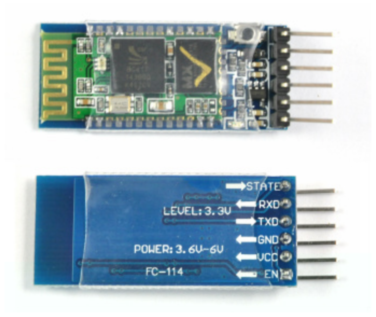


Figure 5: View of the HC-05 surfaces.

3.1 Command and Data Transfer modes

This module has two modes of operation:

- *Command Mode* (or *AT Mode*): used for configuring the module (through AT commands);
- *Data Mode*: user for transmitting and receives data to another Bluetooth module.

3.2 Default settings

The default settings for new modules are:

- Device name: HC-05;
- Password: 1234;
- Baud rate in communication mode: 9600 character per second (but sometimes 38400 character per second);
- Baud rate in *AT Mode*: 38400 character per second;
- State: there are two possible state for this module, master and slave, the second is the default behaviour.

The default device mode is the *Data Mode* and it is used to broadcast data through Bluetooth.

3.3 *AT Mode*

AT command mode allows to interrogate and to modify the previous settings described. Changing the module state, it is possible configuring it to automatically connect to another Bluetooth device, as it will be made in this project.
To use the Command mode it is necessary use an USB to TTL adapter or an opportunely configurated Arduino board. The HC05 should be connected to one of these device according to the following scheme:

HC-05 pin	**For *USB to TTL***	**For *Arduino***
(1) +5V	Vcc	Vcc
(2) GND	GND	GND
(3) Tx	Rx	Rx
(4) Rx	Tx	Tx

3.3.1 Enter in *AT Mode* though EN pin

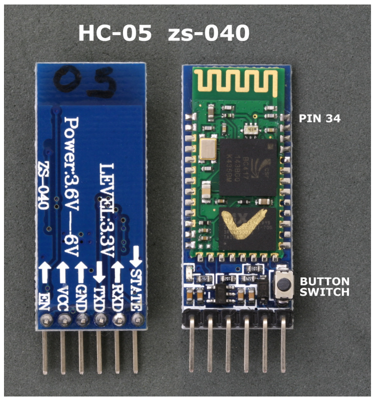


Figure 6: HC-05 module with EN pin and button switch.

To activate *AT Mode* on these HC-05 modules, the pin 34 has to be on power up. For this purpose there is two different ways: if there is a button, if pressed, this will automatically connect the pin 34 to the high power source, else it is possible bring the power to it through a jumper connected to the high source.
To enter on this mode, it is necessary that the power is applied to the module when the pin 34 has an high tension. Once the module has booted, the tension on pin 34 can become high impedance.
To verify if the module is in *AT Mode*, the LED on the HC05 should blink with a frequency of half Hertz.

3.4 Setup the communication

Once the HC-05 is in *AT Mode*, the module can be configured through the *Arduino Serial Module*. The configuration will require that both *NL* and *CR* as end line are enabled and the communication will be kept at 38400 characters per second.
It is necessary set two modules, one for the master role and one for the slave one, since the two modules should be connect each other automatically, without the support of the connected boards. The master will be seek a Bluetooth module with the given address, owned by the slave, and will establish the connection.

3.4.1 Slave configuration

The slave mode will be used on the Arduino:

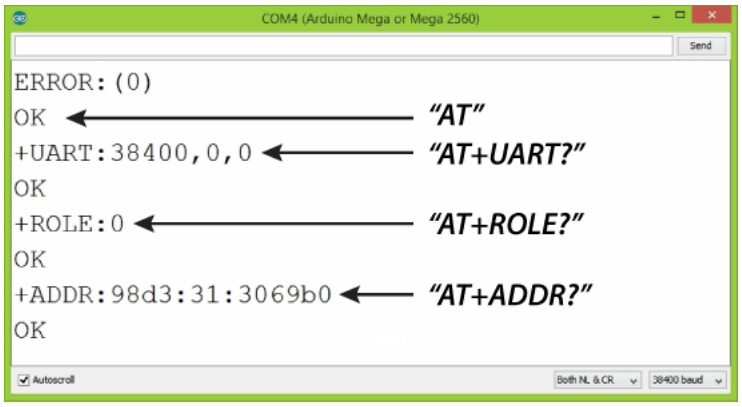
1. Boot the HC05 in *AT Mode* and connect it to the Arduino;

2. To verify if the connection is right, it is possible to type *AT*, which sould get as answer the word *OK*;

3. Typing *AT+UART?*, the module will answer with the actual baud rate, which should be 9600 characters per second, compatible with the Arduino serial port;

4. Typing *AT+ROLE?*, the module should answer a message like *+ROLE=0*, which means that the Bluetooth device is in slave mode;

5. Typing *AT+ADDR?*, the module should answer with its address, like 18:e4:34cldd, and it is necessary to use this to configure the master module.



```
ERROR:(0)
OK
+UART:38400,0,0
OK
+ROLE:0
OK
+ADDR:98d3:31:3069b0
OK
```

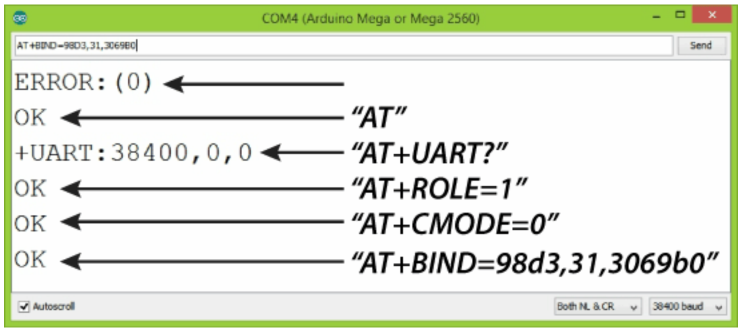
Both NL & CR   38400 baud

Figure 7: Slave configuration of the HC-05 module trough serial monitor.

3.4.2 Master configuration

The master module will be used on the Discovery board. The configuration process is similar to the previous module so here will be exposed only the relevant commands.

1. Boot the HC05 in *AT Mode* and connect it to the Arduino;
2. The STM32F407 uses *19200* baud rate to communicate on USART2, so it is necessary changing the hc-05 baud rate, through the command *AT+UART=19200,0,0*;
3. Typing *AT+ROLE=1*, the serial device will set the Bluetooth module as a master device;
4. Typing *AT+CMODE=0* the serial device will set the connection mode as *fixed address* and, using the *AT+BIND=XX,XXXX,XX* command, putting there the slave address, the master will be set to search this module and connect there (in the address specification it is required using commas, instead of colons).



```
COM4 (Arduino Mega or Mega 2560)
AT+BIND=98D3,31,3069B0

ERROR:(0)
OK              <-- "AT"
+UART:38400,0,0 <-- "AT+UART?"
OK              <-- "AT+ROLE=1"
OK              <-- "AT+CMODE=0"
OK              <-- "AT+BIND=98d3,31,3069b0"
```

Figure 8: Master configuration of the HC-05 module trough serial monitor.

3.5 The communication to the HC05 modules

3.5.1 The transmitter

The code on the Discovery uses the USART2 channel to communicate with the connected HC-05 module.
So, it is necessary link the HC-05 *Rx* pin to Discovery PA2 pin (USART2 *TX*) and HC-05 *Tx* pin to PA3 (USART2 *RX*)
In this project, the Discovery only send strings to Arduino, so the code will only write on the PA2 pin but the implementation regards also the contrary stream for future expansions. To send a string, it is just require to use the *printf* function implemented in the Miosix kernel.

3.5.2 The receiver

The project uses the pin 12 (*TX*) and the pin 11 (*RX*) to communicate with the HC-05 module. To read a string from the HC-05 on Arduino is enough to use the *read()* function of the SoftwareSerial class.

4 Receiver

The receiver is basically a robot driven by the commands from the Discovery board, which communicates through the Bluetooth channel. The receiver is provided by an opportune object to take the data, ignore the eventual dirty characters, and execute what is required.

4.1 Main parts

The robot is divided in three main parts:

- The serial communication: this is the section that is in charge of receiving and sending strings, encapsulating all the code within an object and, consequently, allowing changes and extensions;
- The Executer: this is the object that, given a command stored in a string, looks for a function able to execute it and, in positive case, runs it;
- The controller: this is the part performed by the Arduino *loop* method and provides to invoke the serial communication methods, taking the commands strings and giving them to the executer.

4.2 The serial communication

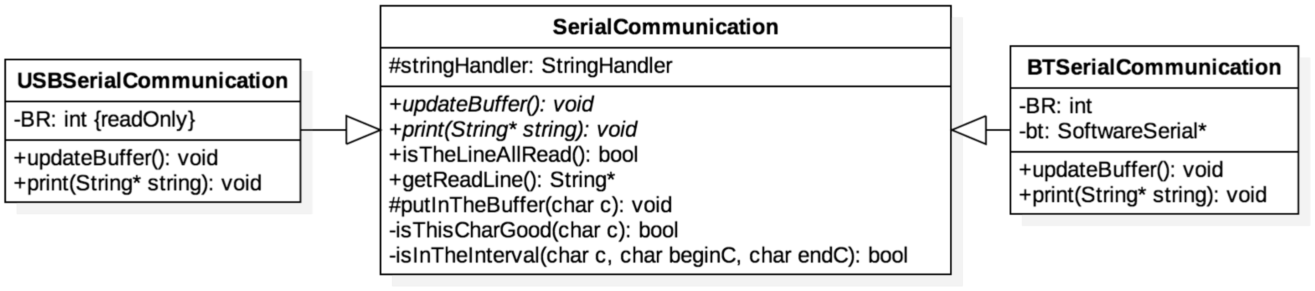

USBSerialCommunication
-BR: int {readOnly}
+updateBuffer(): void
+print(String* string): void
SerialCommunication
#stringHandler: StringHandler
+updateBuffer(): void
+print(String* string): void
+isTheLineAllRead(): bool
+getReadLine(): String*
#putInTheBuffer(char c): void
-isThisCharGood(char c): bool
-isInTheInterval(char c, char beginC, char endC): bool
BTSerialCommunication
-BR: int
-bt: SoftwareSerial*
+updateBuffer(): void
+print(String* string): void

Figure 9: The structure of the connection in the receiver.

The *SerialCommunication* class is a generic structure that takes the characters from a source and converts them in strings. Consequently, this class provides few important methods to improve and to organize this process, used independently by the source nature:

- *updateBuffer*: this method is used to take the available characters from the receiver buffer and put them in a temporary opportune storage object (*StringHandler*). If there is no characters or a line is fully read, this method simply does nothing.

- *isAllTheLineRead*: this methods returns true if a line is fully read, else it returns false.
- *readLine*: this returns in a string all the read characters, independently if a line is fully received, and empties the buffer in *StringHandler*.
- *print*: this method prints the passed string on the connection channel, adding automatically the end line character.

4.2.1 *StringHandler*

All the received characters are temporary stored in an apposite data structure, called *StringHandler*. This class offers some methods to manipulate these characters, allowing to know if a string is fully received and, eventually, withdraw it. It's not necessary that the string is fully received to transform all the received character in a string. After the withdraw method invocation, the object resets itself.

4.2.2 *SerialCommunication* extensions

In the Figure 9, in particular in the parent class, two methods are indicate as virtual, since their implementation depends by the connection type used. Consequently, the *SerialCommunication* class was extended by *BTSerialCommunication* and by *USBSerialCommunication*, which implement these two methods, according to their specification and using proper attributes to perform these actions.
In the communication, the class *USBSerialCommunication* and the *print* method of *BTSerialCommunication* are implemented but never used. They were added for debug features and for future expansion, made easier by this organization.

4.3 The Executer

The *Executer* interprets the received commands by the connection. A command is a string that has, as first character, a proper name and, subsequently, some parameters for that specific command. Its name is used to get, with a constant time through an Hash map, the method to invoke for executing that specific command. This class exposes only a method, the one invoked passing the string as parameter and that researches and executes the opportune function.
This structure allows to extends the robot with future modules, without changing this object but just adding the new function pointers to the map.

4.3.1 The used Hash map

The quoted Hash map is not a class from the library but a proper class implemented in this code. It is not a perfect Hash function since it uses the module operation to compute the indexes but this choice was made to avoid useless computational time wastes. It is however possible change it since it is fully encapsulated in a module.

4.3.2 The used commands

This robot uses the following commands:

- f: to go forward;
- b: to go back;
- l: to go to the left;
- r: to go to the right;
- s: to stop.

The command analyzer is easily extensible and further commands can be implemented, also with parameters.

4.4 The circuit

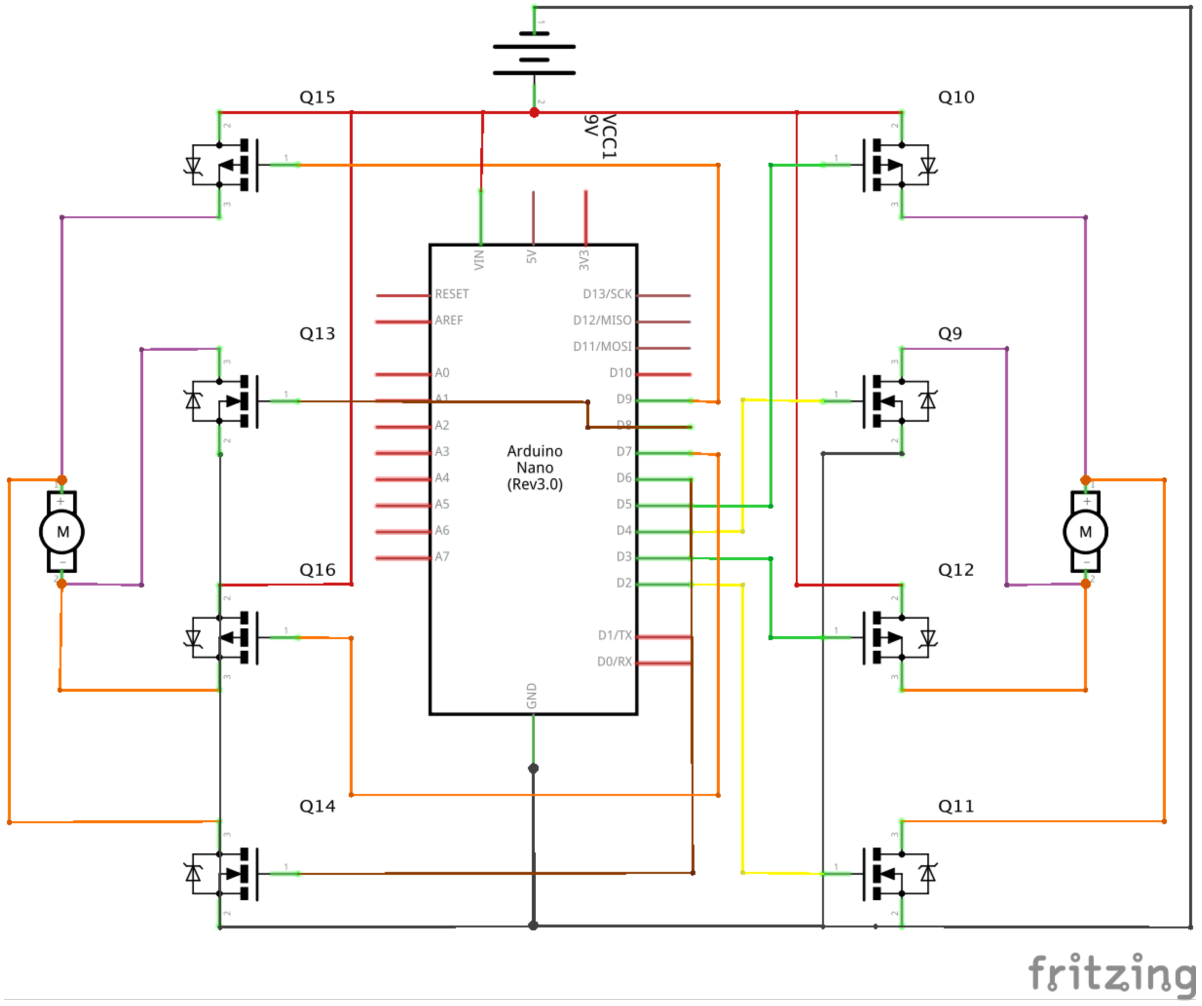


Figure 10: The motors circuit scheme of the robot.

For this robot, is necessary that both the motors rotate in two different directions, consequently, it is implemented two basic *H bridges* to perform this purpose. To avoid the use of inverters, it is necessary uses two pins for each motor direction and so eight Arduino pins are used to handle the movements. In other two pins, the Bluethooth module HC05 text and read pins are connected, despite it isn't shown in the circuit scheme. Also the LEDs and the eventual buzzer aren't shown.

5 Bibliography

1. Miosix: https://miosix.org/;
2. Miosix guides: https://miosix.org/wiki/index.php;
3. Miosix and Discovery examples: http://home.deib.polimi.it/fornacia/doku.php;
4. Arduino documentation: https://www.arduino.cc/reference/en/.