POLITECNICO
MILANO 1863

Embedded Systems
Advanced Operating Systems Project
Academic year 2017/2018

Whistle Robot

Version 0.0
??/?/2018

Borgo Daniele 894110
Fusca Yuri 898602
Indirli Fabrizio 899892

Advisor : Federico Terraneo
Professor: William Fornaciari

Contents

1 Introduction

The main goal that this project aims is to design a game where the Discovery microphone is used to control a robot, through a whistle. In particular, this board, after calculating the sound frequency, sends to the robot a related command for choosing the direction to take. The game consists in a race with obstacles where the winner will be the one who complete the path before the others.

1.1 Problem statement

The game consists in a race between two or more robots, where each player controls one of the machines through whistles using the related Discovery board. The most skilled player in the control of the robot will be the one who arrives first at the end of the path, gaining the victory. All impacts to the other player are allowed and warmly suggested.
A specific movement of the robot will correspond to a precise frequency range:

- Forward movement
- Backward movement
- Left movement
- Right movement

1.2 Summary of the work

The code is organized in three main sections, in order to improve the maintainability and the creation process:

- *The Transmitter*: this part is represented by the Discovery board and it converts the perceived sounds in numerical frequencies. After that, the board picks the expected command for that value and sends it to the robot.
- *The Communication*: this is the part that transfers the command from the Discovery to the Arduino through a Bluetooth channel.
- *The Receiver*: this is the real robot, which executes the received commands, changing its state.

2 Transmitter

The transmitter is the part that recognizes sounds (through a microphone) and produces commands for the robot. 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 to give to the robot.

2.1 The MP45DT02 microphone

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

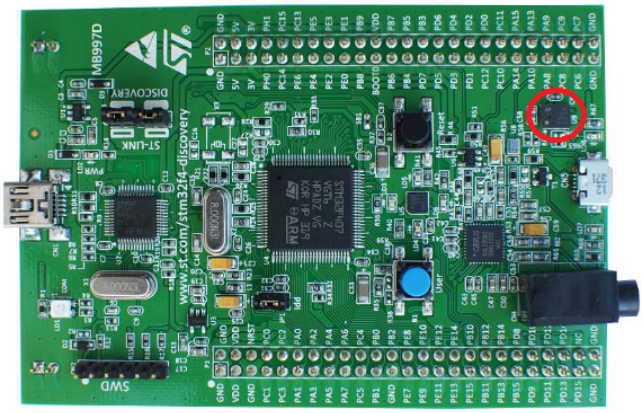

Figure 1: View of the upper surface of the board, with the microphone highlighted in the circle.

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.

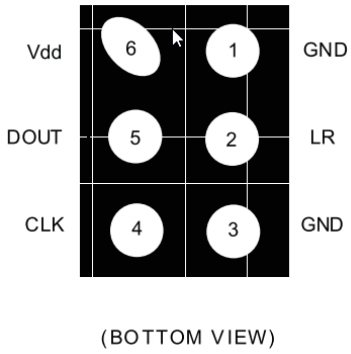

Figure 2: Pins on the bottom of the microphone.

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 sixteen (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 ivoke 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 sends the commands to the Arduino board. This object avoids to repeatedly send the same command, sending only the first of an all equal string series.

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()*.

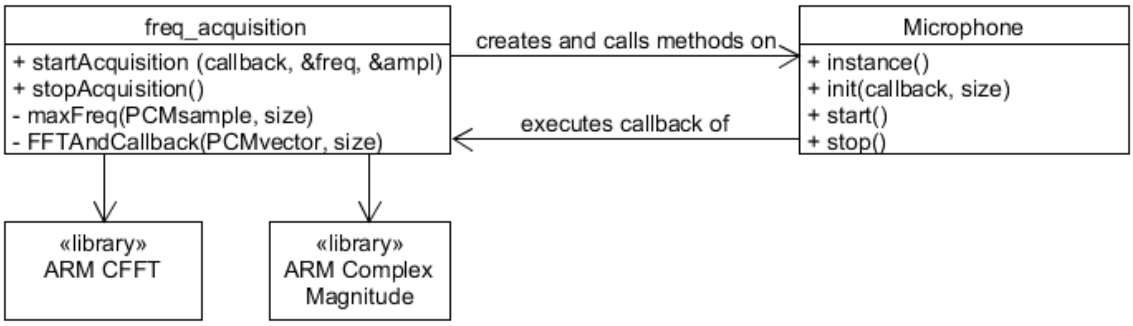

Figure 3: UML schema of the interaction between the modules in frequency recognition.

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 convey the commands from the transmitter to the receiver, a wireless connection is needed, in this case implemented through two HC-05 modules. These boards are, in technical terms, Bluetooth SPP (Serial Port Protocol) modules, used to abstract the wireless connection implementation.

Figure 4: 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*: used for transmitting and receiving data to/from 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 characters per second (but sometimes 38400 characters per second);
- Baud rate in *AT Mode*: 38400 characters per second;
- State: there are two possible states 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 to configure it to automatically connect to another Bluetooth device, as it will be made in this project.
To use the Command mode, an USB to TTL adapter or an opportunely configured Arduino board is needed. The HC05 should be connected to one of these devices 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* through EN pin

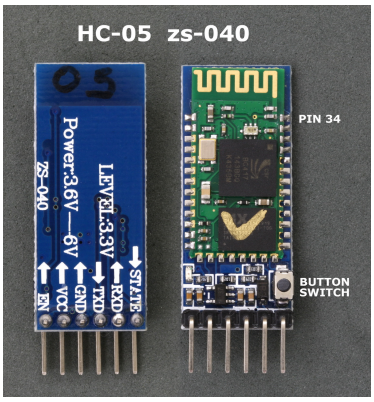

Figure 5: HC-05 module with EN pin and button switch.

To activate *AT Mode* on these HC-05 modules, the pin 34 has to be high on power up. For this purpose there are two different approaches:

- if there is a button on the module, press it: this will automatically connect the pin 34 to the high power source
- if not, manually connect pin 34 to a 5V voltage source through a jumper wire.

To enter this mode, the pin 34 must be connected to at least 4V when powering on the module. Once the module has booted, the voltage on pin 34 can become high impedance.
To verify if the module is in *AT Mode*, the LED on the HC05 should blink with halved frequency.

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 to set the two modules, one for the master role and one for the slave one, since the two modules should connect to each other automatically, without the support of the boards. The master will seek a Bluetooth module (the slave) whose address is the given one, 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 should return *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.

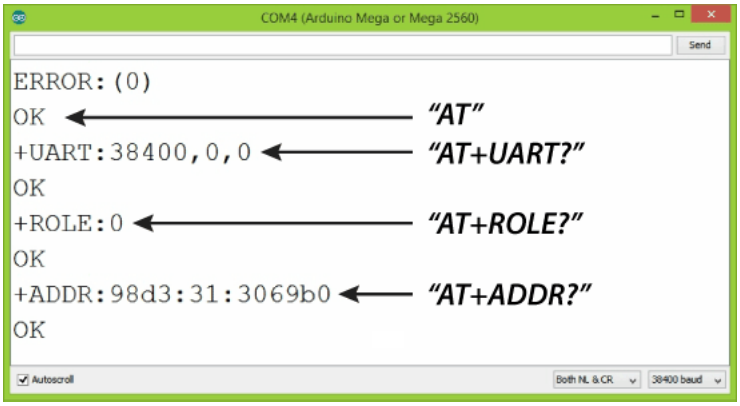

Figure 6: Slave configuration of the HC-05 module trough serial monitor.

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 only the relevant commands will be explained.

1. Boot the HC05 in *AT Mode* and connect it to the Arduino;
2. The STM32F407 uses *9600* baud rate to communicate on USART2, so it is necessary to change the hc-05 baud rate, through the command *AT+UART=9600,0,0*;
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).

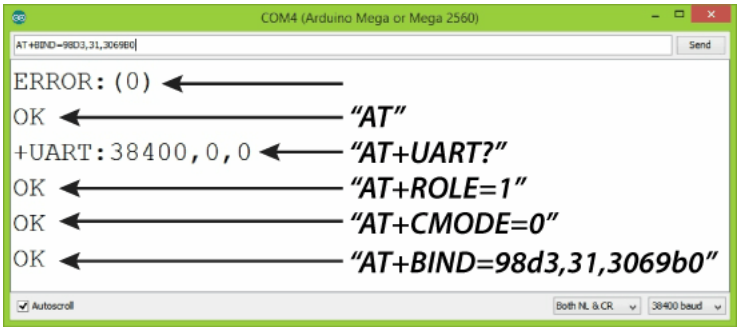

Figure 7: Master configuration of the HC-05 module trough serial monitor.

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 to 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 implemented by an opportune object that takes the data, ignores the eventual dirty characters and executes the bound function.

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* metho; it invokes the serial communication methods, taking the commands strings and giving them to the executer.

4.2 The serial communication

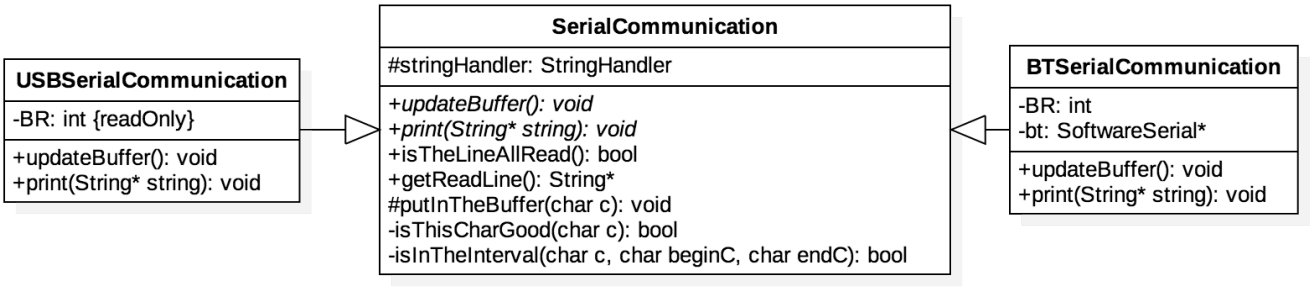

Figure 8: The structure of the connection in the receiver.

The *SerialCommunication* class is a generic structure that acquires 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 are no characters or a line is fully read, this method simply does nothing.

- *isAllTheLineRead*: this method 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 temporarily 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 characters in a string. After the withdraw method invocation, the object resets itself.

4.2.2 *SerialCommunication* extensions

In the Figure 8, in particular in the parent class, two methods are indicated 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.

4.2.3 Optimization

Since this control requires a lot of milliseconds to perform its analysis, in the receiver it is implementer a more optimized structure to avoid this problem, in case of the need of more *Real Time* performances. This structure forbids the parameters use in the connection channel, aiming to execute the command as soon as possible, reducing the buffer queue and gaining smaller latencies.
This structure replaces the entire connection described with a simple object that return characters, instead of strings, taken directly by the serial channel. These characters represent the received commands without parameters and they are used by an opportune method of the *Executer* object to seek and execute the opportune command. Since the architecture doesn't allow parameters, all the invoked functions received an empty string as parameter.
This structure should be used when the commands stream by the Discovery is very high, consequently, the sound is very various. The optimization, on other words, empties the receiver buffer faster than the other structure.

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 seeks 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

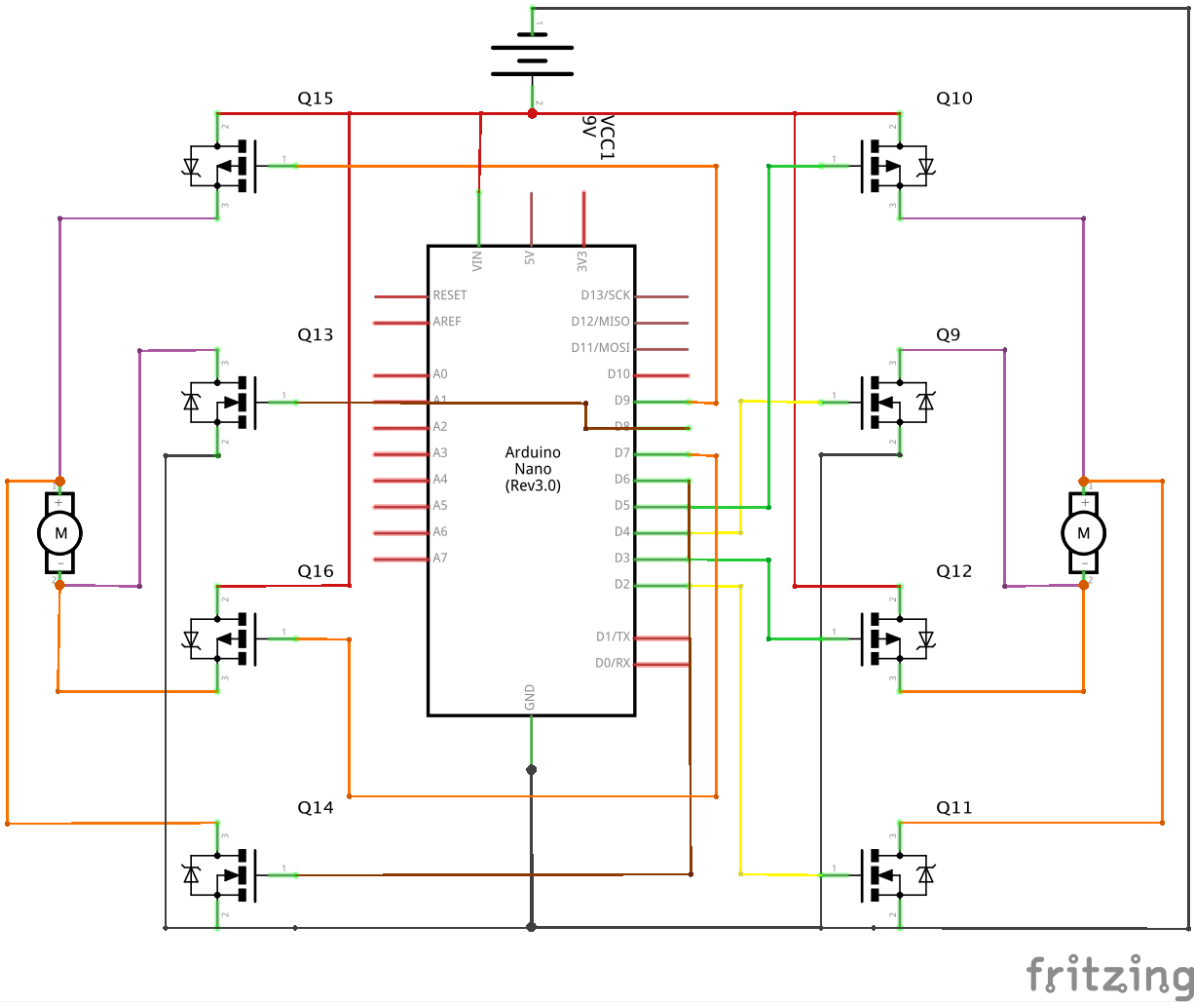

Figure 9: The motors circuit scheme of the robot.

For this robot, is necessary that both the motors rotate in two different directions, consequently, two basic *H bridges* have been implemented to achieve this purpose. To avoid the use of inverters, it is necessary to use two pins for each motor direction, hence 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/.
5. Lorenzo Binosi and Matheus Fim's Guitar Tuner Project: https://github.com/LorenzoBinosi/AOSProject?files=1
6. Tilen Majerle's tutorial *STM32F4 FFT Example* on *stm32f4-discovery.net*: https://stm32f4-discovery.net/2014/10/stm32f4-fft-example/
7. *Spectrum analyzer with FFT - example of application on STM32F4DISCOVERY* article on *stm32.eu*: https://stm32.eu/2014/01/09/analizator-widma-z-fft-przyklad-aplikacji-na-stm32f4discovery/
8. AT Command mode tutorial: http://www.martyncurrey.com/arduino-with-hc-05-bluetooth-module-at-mode/;