POLITECNICO
MILANO 1863

Embedded Systems
Advanced Operating Systems Project

Whistle Robot

Version 0.0
??/?/2018

Borgo Daniele 894110
Fusca Yuri ??????
Indirli Fabrizio 899892

Contents

1 Introduction **1**
- 1.1 Purpose . . . 1
- 1.2 Description of the robot . . . 1
- 1.3 Game logic . . . 1
- 1.4 Code structure . . . 1

2 Controller **3**

3 Transmitter **4**
- 3.1 The MP45DT02 microphone . . . 4
- 3.2 Overview on sound acquisition and processing . . . 5
  - 3.2.1 Microphone initialization . . . 6
  - 3.2.2 Sound acquisition and processing . . . 6
- 3.3 The *freq_recognition* library . . . 7
  - 3.3.1 The Microphone.cpp driver . . . 7
  - 3.3.2 Interface of the library . . . 7
  - 3.3.3 How it works . . . 7
  - 3.3.4 Usage of the library . . . 8
- 3.4 Displaying the recognized frequency . . . 9
  - 3.4.1 Connections . . . 9
  - 3.4.2 Driver and usage . . . 10

4 Wireless Communication **12**
- 4.1 COMMAND and DATA TRANSFER MODES . . . 12
  - 4.1.1 How to enter command mode if your HC-05 has KEY pin 13
  - 4.1.2 How to enter command mode if your HC-05 has EN pin instead of KEY pin . . . 14
- 4.2 Setup the communication . . . 16
- 4.3 HOW TO PAIR THE TWO MODULES . . . 16
  - 4.3.1 Slave configuration (this HC-05 module will be used on the Arduino Board) . . . 16
  - 4.3.2 Master configuration (this HC-05 module will be used on the STM32F407 Discovery Board) . . . 17
- 4.4 STM32F407 CODE (Transmitter) . . . 18
- 4.5 ARDUINO CODE (Receiver) . . . 18

5 Receiver **19**
- 5.1 Main parts . . . 19
- 5.2 The serial communication . . . 19
  - 5.2.1 *StringHandler* . . . 20
  - 5.2.2 *SerialCommunication* extensions . . . 20
- 5.3 The Executer . . . 20
  - 5.3.1 The used Hash map . . . 20
  - 5.3.2 The used commands . . . 21

5.4 The circuit 21

6 Bibliography **23**

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.

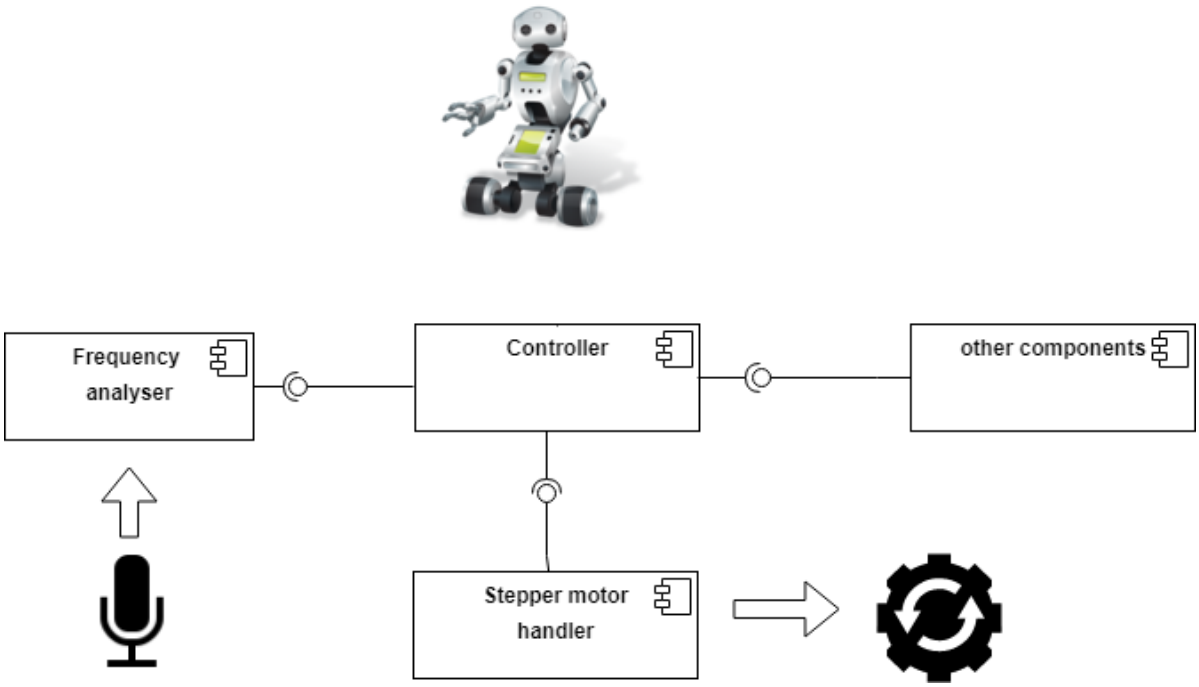

Figure 1: This is a general overview of the system.

2 Controller

3 Transmitter

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.

3.1 The MP45DT02 microphone

The MP45DT02-M is a compact, low-power, topport, omnidirectional, digital MEMS microphone. It's soldered on the top of the STM32F407G-DISC1 board, in the bottom-right corner.

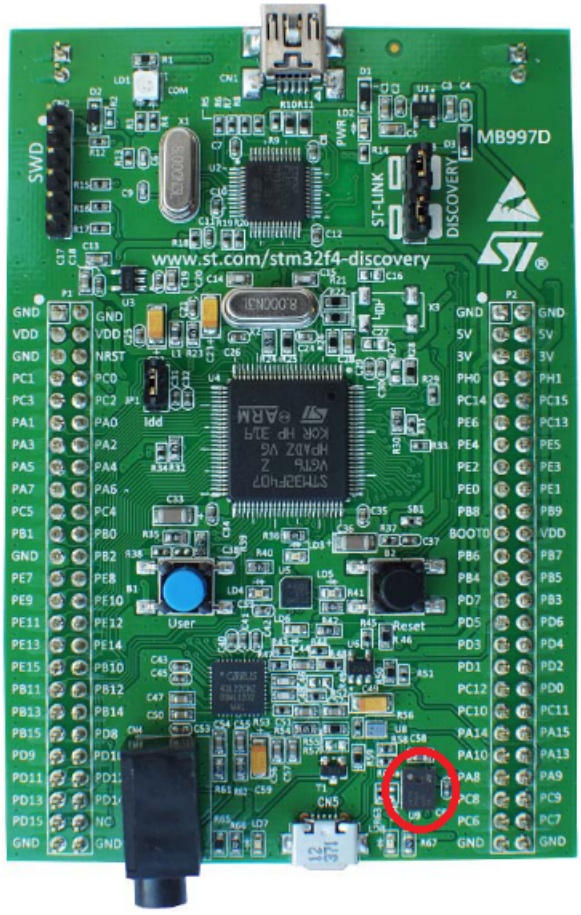

Figure 2: View of the upper surface of the board, with the microphone highlighted in the circle.

In its bottom surface, it has six pins:

1. **GND**: Connected to the GND of the board.
2. **LR** - Channel selection: used because 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.

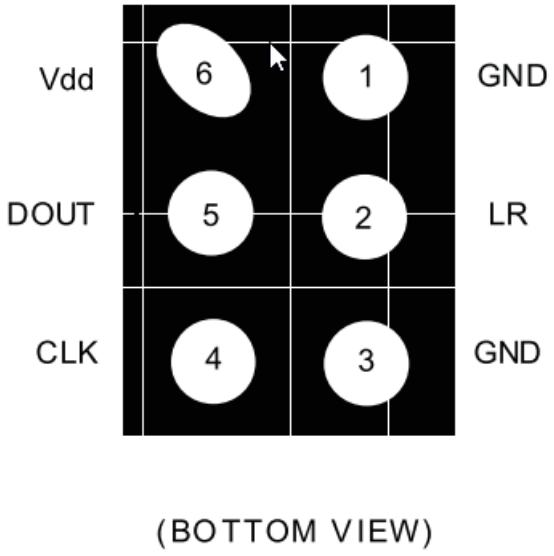

Figure 3: Pins on the bottom of the microphone.

3.2 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.

3.2.1 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

3.2.2 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 command are then sent via Bluetooth to an Arduino board that drives the engines.

3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 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()*.

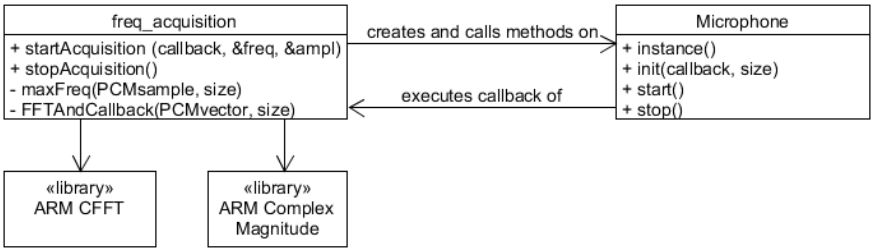

Figure 4: UML schema of the interaction between the modules in frequency recognition.

3.3.4 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.

3.4 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).

3.4.1 Connections

The display has sixteen pins that are connected in this way:

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

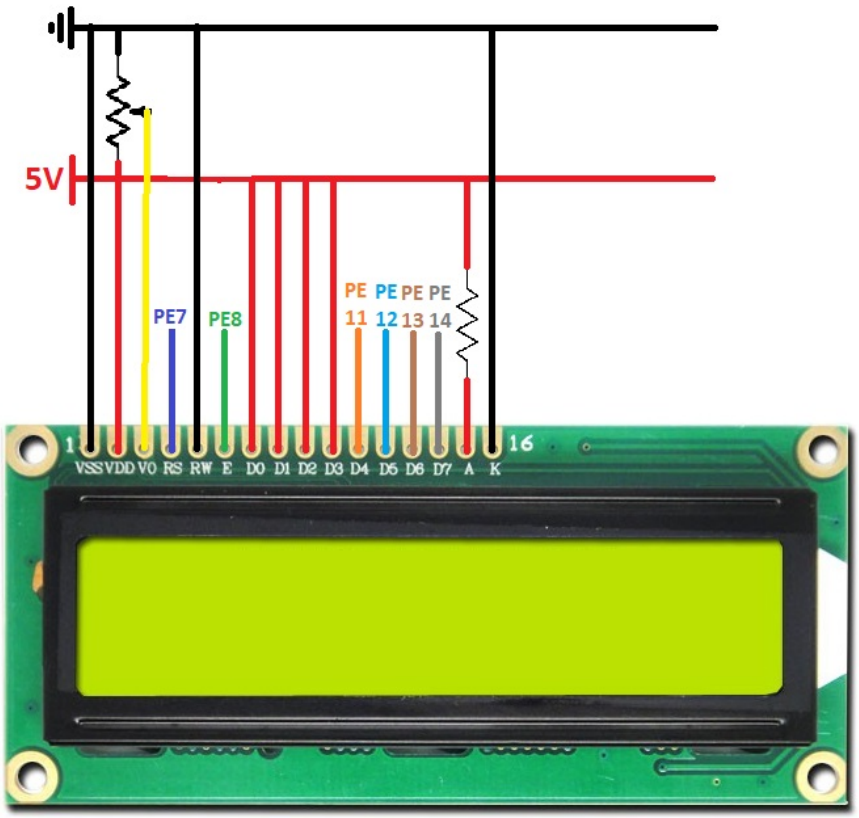

Figure 5: Schema of the display's connections.

3.4.2 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 with 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.
Then, we can use the methods of the Lcd44780 object to print on the screen:

- *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.

In this project, the *main* function instantiates a Lcd44780 object called *display*, and calls its methods *clear()* and *go(0,0)* to initialize it. Then, each time the callback function is called from the frequency recognition library, it uses the *printf()* method to print the recognized frequency and the associated command; if the volume of the acquired sample is too low, *Volume too low* will be printed on the screen instead of the frequency.

Figure 6: Display showing the frequency and the associated command.

4 Wireless Communication

For this project, we have decided to allow communication between STM32F407 and Arduino board trough two HC-05 modules.
HC-05 module is an easy to use Bluetooth SPP (Serial Port Protocol) module, designed for transparent wireless serial connection setup. The HC-05 Bluetooth Module can be used in a Master or Slave configuration, making it a great solution for wireless communication.

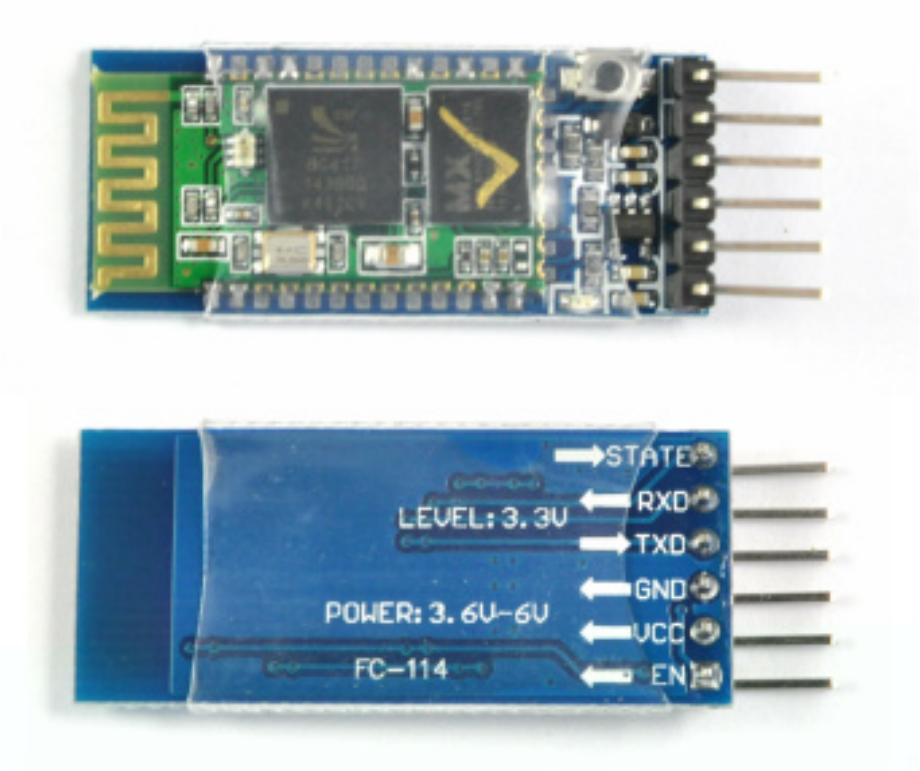

Figure 7: View of the surfaces of the HC-05 module.

4.1 COMMAND and DATA TRANSFER MODES

The module has two modes of operation: **Command Mode** where we can send AT commands to it, and **Data Mode** where it transmits and receives data to another Bluetooth module.

Default Settings
The default settings for new modules are:

- Name = HC-05
- Password = 1234
- Baud rate in communication mode = 9600*
- Baud rate in AT/Command mode = 38400

*sometimes 38400

The default mode is DATA Mode, and it is used to transmit data over Bluetooth communication.
AT command mode allows you to interrogate the Bluetooth module and to change some of the settings: like the name, the baud rate, whether or not it operates in slave mode or master mode.
When used as a master device AT commands allow you to connect to other Bluetooth slave devices.
For the project, we need to change some of the configuration setup values in AT command mode.

There are two ways to operate in Command Mode: using a USB to TTL adapter, or using an Arduino board.
Link the HC-05 following this scheme (dont plug +5V pin for now)

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

4.1.1 How to enter command mode if your HC-05 has KEY pin

HC-05 pin	**Connected to *USB to TTL***	**Or connected to *Arduino***
Key	Vcc	3.3V

KEY: This pin has to be pulled high to enter AT mode.

4.1.2 How to enter command mode if your HC-05 has EN pin instead of KEY pin

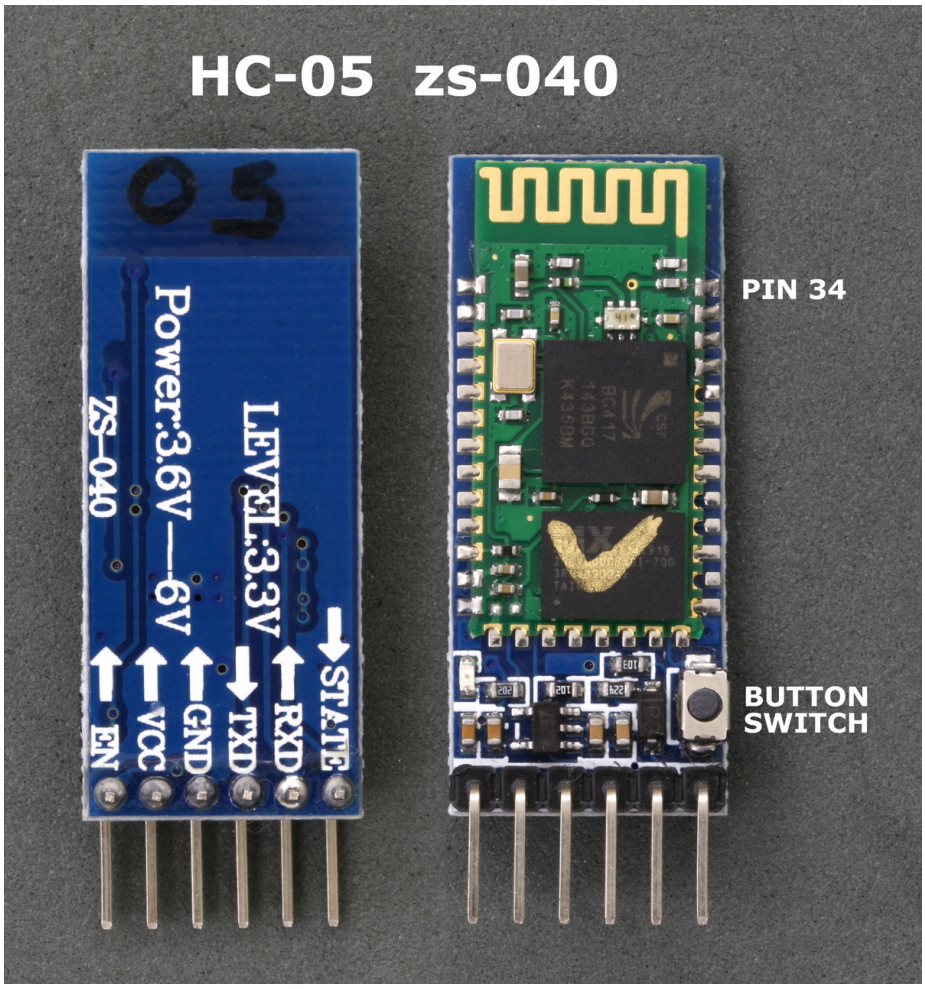

Figure 8: HC-05 module with EN pin and switch button.

To activate AT mode on these HC-05 modules, pin 34 needs to be HIGH on power up. The small push button switch connects pin 34 to +3.3v so we can either:

- connect pin 34 directly to +3v3v and power on, or
- hold the button switch closed when starting the module.

METHOD 1 - Using the button switch to enter AT command mode:

1. remove power from the module
2. hold the small button switch closed while powering on the module.
3. press and hold the button switch.
4. while still holding the button switch closed, apply power.
5. when you see the LED come on you can release the button switch.

If the Bluetooth module led is flashing every 2 seconds (slower than usual) that means that we have successfully entered in the AT command mode.
METHOD 2 - Using pin 34 to enter full AT command mode:

1. remove power from the module
2. make a connection between pin 34 and +3.3v
3. reapply power.

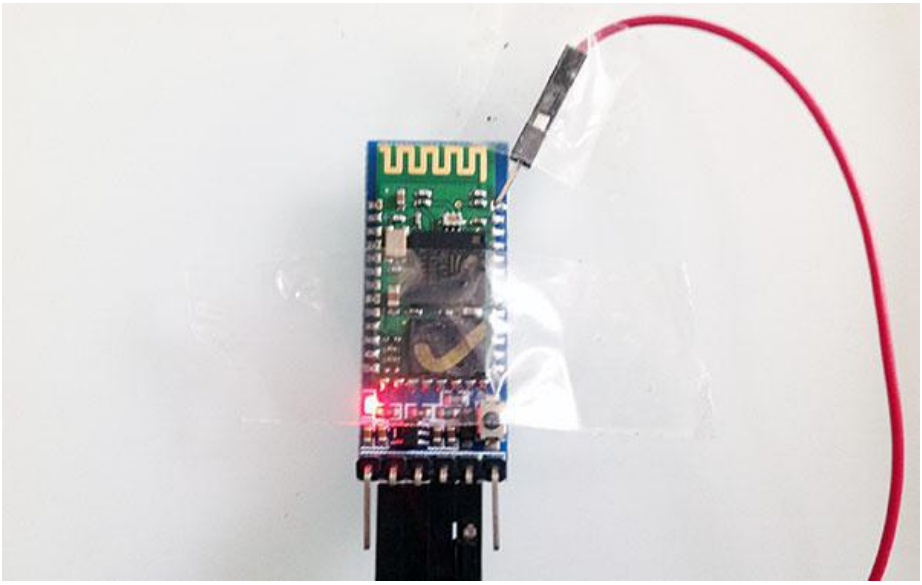

Figure 9: Example of connection bewteen pin 34 and +3.3v.

If the Bluetooth module led is flashing every 2 seconds (slower than usual) that means that we have successfully entered in the AT command mode. prova
prova
prova
prova

4.2 Setup the communication

Final scheme:

STM32F407 → HC-05 (master)  HC-05 (slave) → Arduino

Once the HC-05 is in Command mode, we can use *Arduino Serial Monitor* to set the HC-05 module.
After we run the Serial Monitor, we need to select *Both NL and CR*, as well as, *38400 baud* which is the default baud rate in command mode. Now, we are ready to send commands.

4.3 HOW TO PAIR THE TWO MODULES

4.3.1 Slave configuration (this HC-05 module will be used on the Arduino Board)

If we type just *AT* which is a test command we should get back the message *OK*.
Then if we type *AT+UART?* we should get back the massage that shows the default baud rate which is *9600*. (**REMEMBER, we will use the same baud rate on the Arduino board)**
Then if we type *AT+ROLE?* we will get back a massage *+ROLE=0* which means that the Bluetooth device is in slave mode. If we type *AT+ADDR?* we will get back the address of the Bluetooth module and it should looks something like this: 18:e4:34cldd.
Now we need to write down this address, as we will need it when configuring the master device.

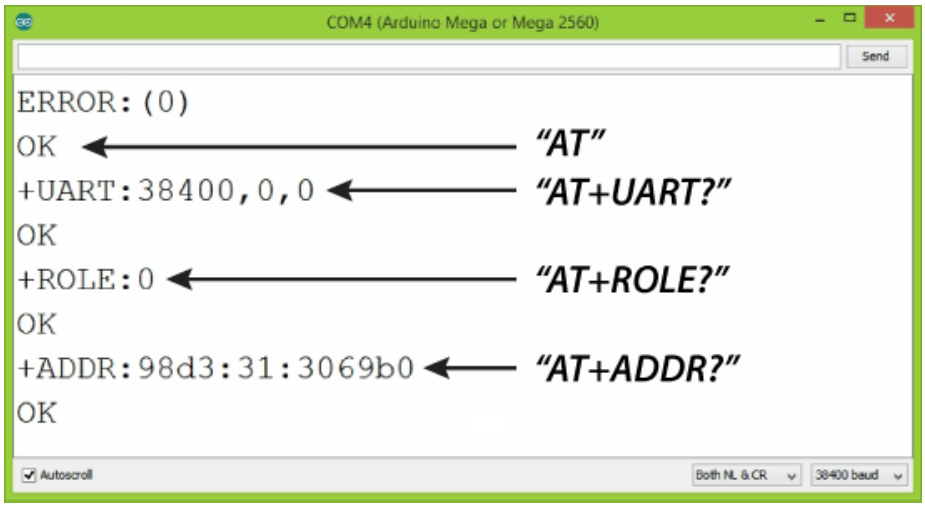

Figure 10: Slave configuration of the HC-05 module trough serial monitor.

4.3.2 Master configuration (this HC-05 module will be used on the STM32F407 Discovery Board)

The STM32F407 uses *19200* baud rate to communicate on USART2, so we need to change the hc-05 baud rate, in addition to pair the master with his slave.
To speedup the baudrate to *19200 bauds*, we have to type *AT+UART=19200,0,0*. Then by typing *AT+ROLE=1* we will set the Bluetooth module as a master device. After this using the *AT+CMODE=0* we will set the connect mode to fixed address and using the *AT+BIND=XX,XXXX,XX* command we will set the address of the slave device that we previously wrote down.
(Use commas, instead of colons!)
AT+BIND=18,e4,34cldd in our case.

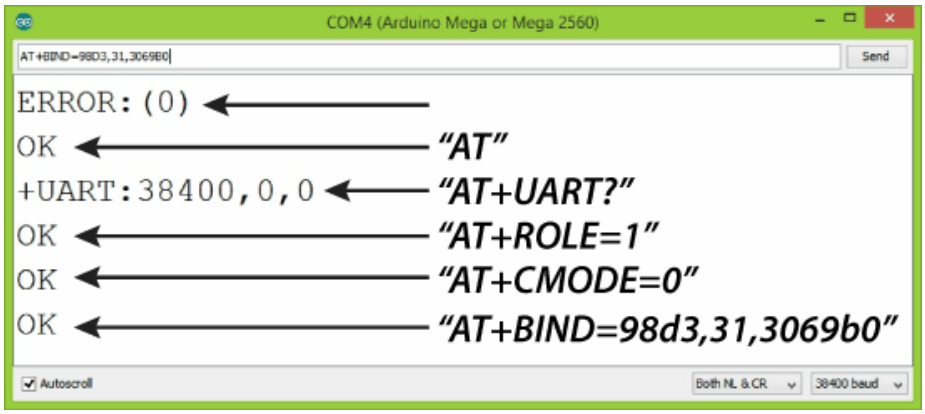

Figure 11: Master configuration of the HC-05 module trough serial monitor.

4.4 STM32F407 CODE (Transmitter)

The code loaded on the STM32F407 uses the USART2 channel to communicate with the HC-05 module.
So, we should link the HC-05 Rx pin to **PA2 pin (USART2 TX)** and HC-05 Tx pin to **PA3 (USART2 RX)**
In our project, we only send strings from SMT32F407 to Arduino (not vice versa), so the code will only write on the PA2 pin. (It is not necessary to link HC-05 Tx to PA3 pin for our purpose)
To send a string, we just need to use the printf function implemented in the miosix kernel.

4.5 ARDUINO CODE (Receiver)

ArduinoBluetoothTest.txt contains a sketch with allow us to send and receive strings trough Bluetooth communication.
We decided to use **PIN 12 (TX)** and **PIN 11 (RX)** to communicate with the HC-05 module.
So, we should link the HC-05 Rx pin to PIN 12 and HC-05 Tx pin to PIN 11.
In our project we only send strings from SMT32F407 to Arduino (not vice versa), so the code will only read from PIN 11. (Its not necessary to link HC-05 Rx to PIN 12 pin for our purpose)
To read a string from the HC-05 on Arduino is enough to use the *read()* function of the SoftwareSerial class.

5 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.

5.1 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.

5.2 The serial communication

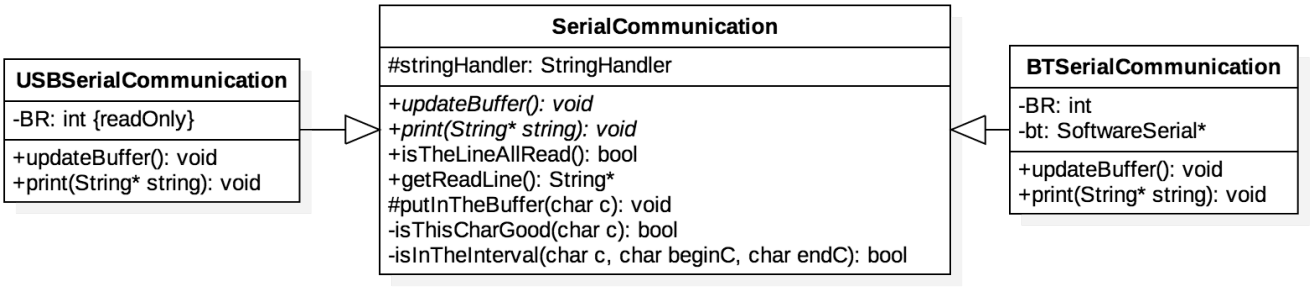

Figure 12: The structure of the connection in the receiver.

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.

5.2.1 *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.

5.2.2 *SerialCommunication* extensions

In the Figure 12, 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.

5.3 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.

5.3.1 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.

5.3.2 The used commands

This robot uses the following commands: ???

5.4 The circuit

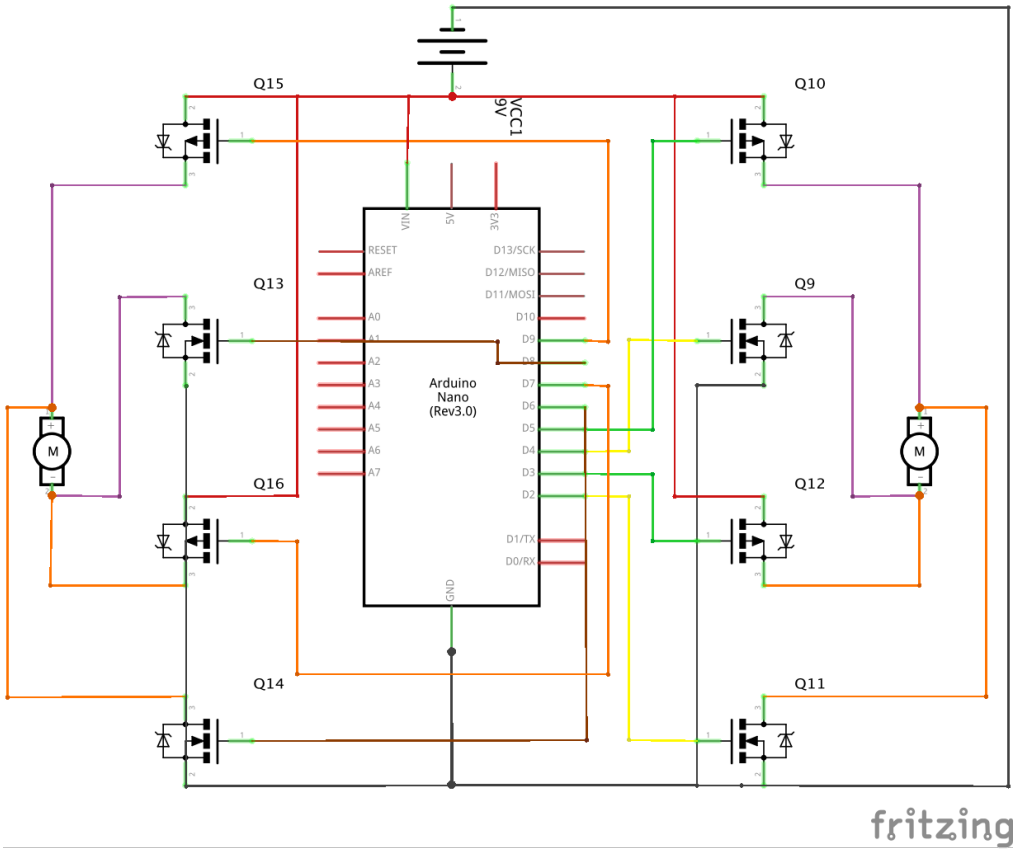

Figure 13: The motors circuit scheme of the robot.

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 con-

nected, despite it isn't shown in the circuit scheme. Also the LEDs and the eventual buzzer aren't shown.

6 Bibliography