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 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.

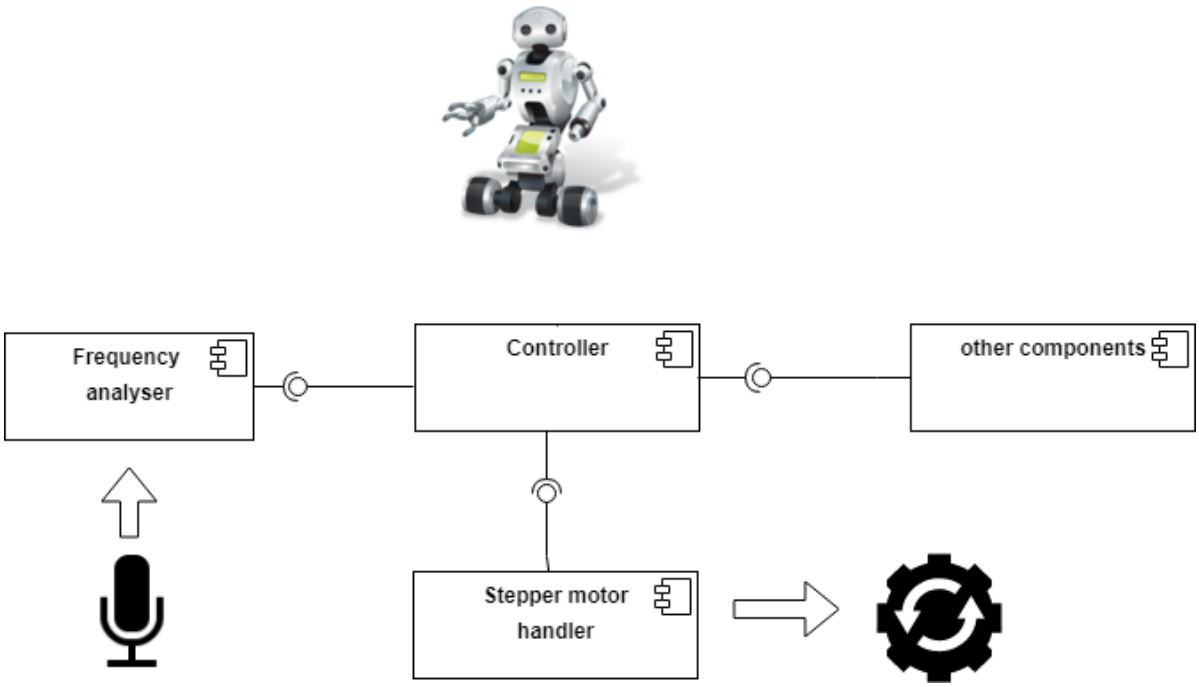

Figure 1: This is a general overview of the system.

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, we will use these devices 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.

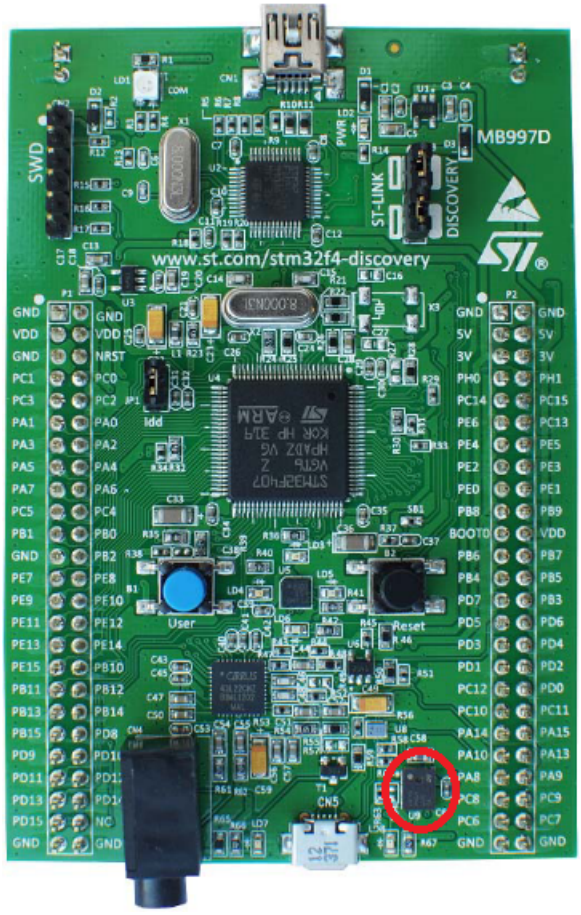

Figure 2: View of the upper surface of the board, with the microphone highlighted in the circle.

In its bottom surface, it has 6 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) 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.

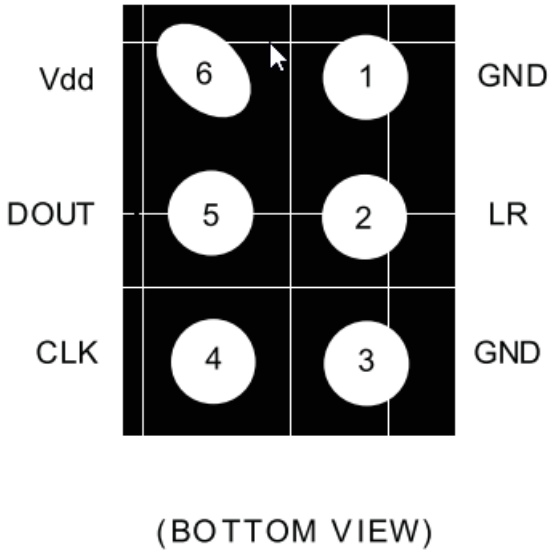

Figure 3: Pins on the bottom of the microphone.

3.2 Overview on sound acquisition and processing

The sound is acquired by the microphone, and the communication with it happens 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 format: this is done by doing **CIC filtering**, 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 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 16 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 16
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 our 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 our project, we have also made some little changes 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()*

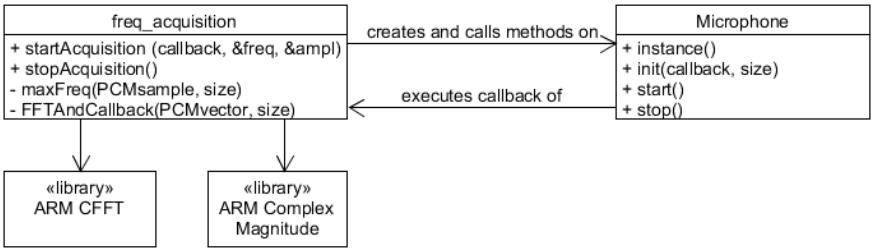

Figure 4: UML schema of the interaction between the modules in frequency recognition.

3.3.4 Usage of the library

The usage of the library is very simple:

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 your 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 display** with 2 rows and 16 columns, that is used to print the currently recognized frequency and the associated command (if any).

3.4.1 Connections

The display has 16 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

-

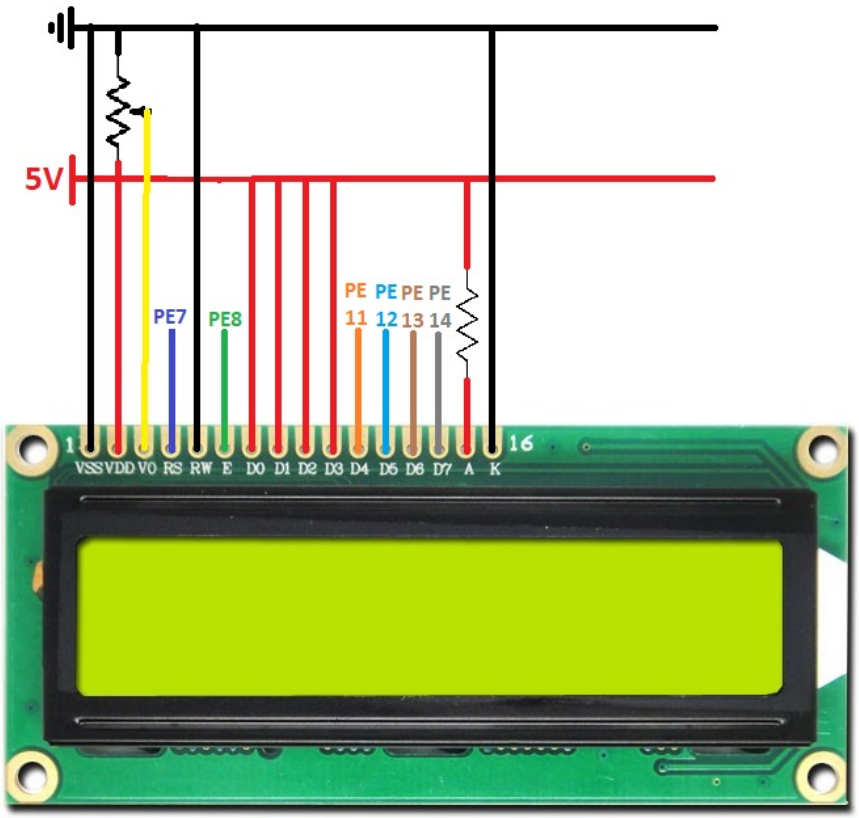

Figure 5: Schema of the display's connections.

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

For more information, read the official tutorial on the Miosix website:
https://miosix.org/wiki/index.php?title=HD44780_LCD_tutorial

In our 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

5 Receiver

The receiver is basically a robot driven by the Discovery board, which sends the opportune commands 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.
The robot is made by three main parts:

- The serial communication: this is the section that is in charge of receiving and sending commands, encapsulating all the code within an object and, consequently, allowing changes and extensions;
- The Executer: this is the object that, given a string, looks for a function able to execute it and 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.

5.1 The Serial Communication

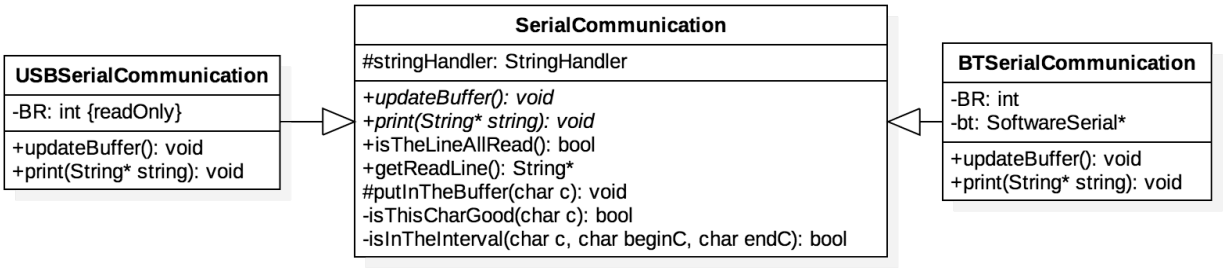

Figure 7: The structure of the connection in the receiver.

The *SerialCommunication* can be seen as a temporary buffer where the character are provisionally stored until an end line token is received. For this reason this object provides few important method useful to this process:

- *updateBuffer*: this method is used to move the available characters from the receiver buffer to this object one. If there is no character or a line is fully read, this method simply does nothing.
- *isAllTheLineRead* and *readLine*: these methods are used to know if a line is fully read and, in positive case, converting all the read characters in a string, emptying the buffer.
- *print*: this method is basically used to print something on the connection channel.

All the received characters are temporary stored in an apposite data structure, called *StringHandler*. This encapsulates all the related methods, like the insertion and the string casting.
In the Figure 7, 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 implemented for debug features and for future expansion, made easier by this organization.

5.2 The Executer

This object allows to interpret the received commands. 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 to execute that specific command. This structure allows to extends the robot with future module, without changing this object but just adding the new function pointers to the map.
For instance, this robot uses the following commands: ???

5.3 The circuit

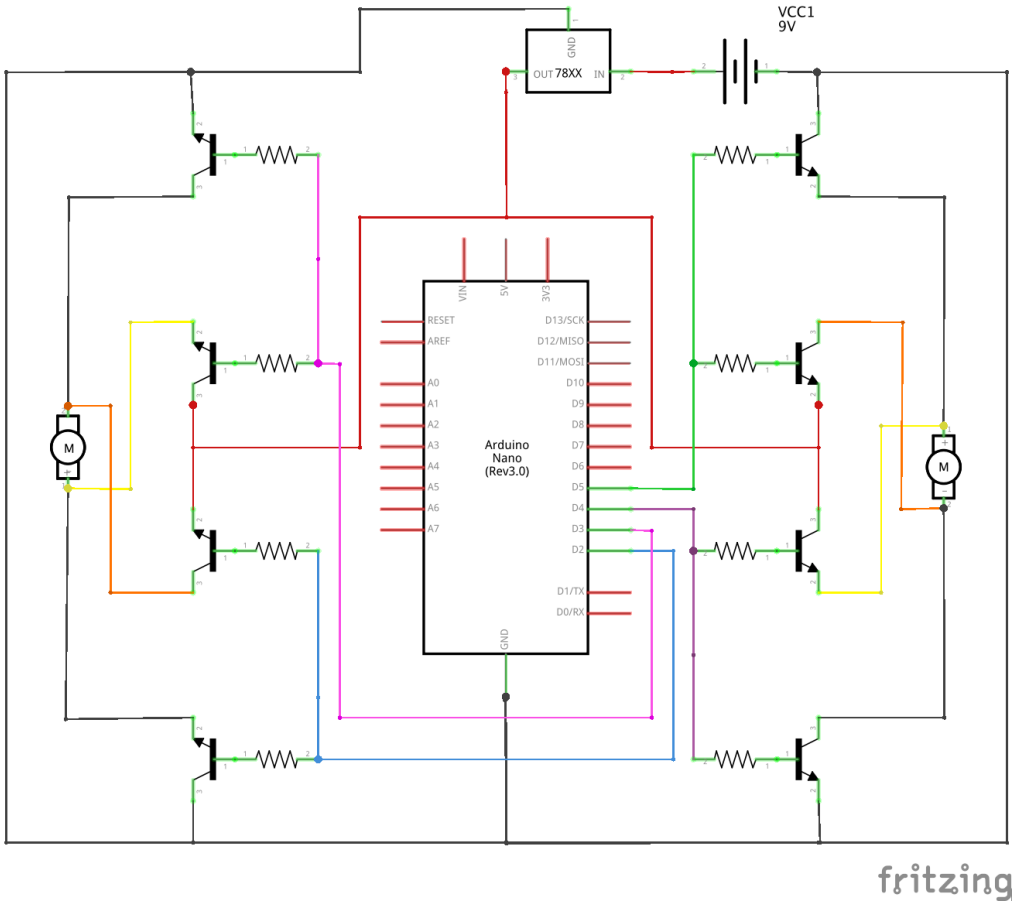

Figure 8: The circuit scheme of the robot. The resistors are all of one KOhm.

This configuration uses eight BJTs to allow the motors to rotate in two different directions, allowing more moves for the robot. The voltage that they receive is around three volts.

6 Bibliography