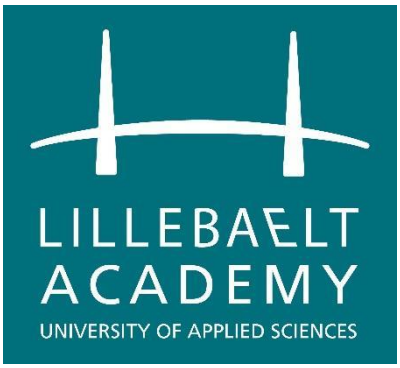
IT Technology
Remote voice control of a system of robot vehicles
Project Report



LILLEBAELT ACADEMY
UNIVERSITY OF APPLIED SCIENCE

Authors
Dainius Čeliauskas
Juan Mohamad

dain0084@edu.eal.dk
juan0314@edu.eal.dk

Sunday, June 3, 2018

Abstract

This document describes a cheap, open source solution for controlling devices with voice commands and receiving measurement data, either locally or through the Internet.

Table of Contents

Table of Figures

I. Introduction

1. The Issue

Voice command devices are becoming more popular as time goes on – the ability to replace buttons, switches and dials with a hands-free user interface that is usable by almost everybody is quite appealing. In fact, voice command research is reported to be an over a billion dollar industry, with tech companies such as Google, Apple and Microsoft creating and selling their own speech recognition engines.

The biggest problem with these systems is their high cost to license and implement in third-party commercial products, as well as their tendency to be closed-source and therefore difficult to modify to suit more specific needs.

Consequently, while voice command devices are becoming more and more important today, it is getting harder to create custom cost-efficient and open source speech recognition implementations.

2. The Approach

The need for a customizable and cheap solution to speech recognition has sparked the creation and development for an alternative to current voice command implementations on the market. The end goal of this project is to develop and demonstrate a system that accepts voice commands on user input and execute dynamic commands based on them, all while being easy to set up and modify.

The system also combines voice control with the ability to remotely communicate with each other using lightweight messaging protocols. It makes it able to, for example, send measurements between devices periodically and even send voice commands over the Internet from one device to another. Its design is flexible, allowing it to be configured and augmented even further.

3. Description and purpose

The project is a system consisting of multiple robot cars controlled by Raspberry Pi computers and a separate Raspberry Pi that is capable of remotely connecting to the robots, called the “hub” device. The robots can control the speed and directions of the controlled motors using voice commands, either from a microphone connected to them or from the “hub” device. The robots can be completely separated from any power outlets or stationary power sources – if the Pi can get 5V DC from an external power source, it is capable of functioning.

A PC can be used to connect to any of the devices. It can read the measurements sent to the “hub” device and the voice commands sent by the user.

dain0084@edu.eal.dk
juan0314@edu.eal.dk

Some of the project's features include:

- Using voice commands to control the speed and direction of the robot car
- Sending periodic measurements of its current RPM (rotations per minute) to an external "hub" device
- Using the external "hub" device to send voice commands remotely to any of the robot cars
- Offline functionality – the speech recognition software does not depend on any online servers for its speech processing, everything is done locally

The results of the project can have several potential applications: supplementation or replacement of existing user interfaces, remote control of devices or helping create solutions for operating wheelchairs hands-free for paralyzed individuals, for instance.

4. Aims and objectives

Primary objective: build a voice recognition controlled motor set capable of receiving commands locally and remotely.

Secondary objectives:

1: Build a program/script that can recognize spoken keywords as input and print out a message or blink an LED.

2: Control a set of motors (make it go forward or backward, turn left or right) with the voice recognition program.

3: Build a device that can periodically get a value; send the value through MQTT to another device.

4: Build multiple sets of controllable motors and a "hub" device; send measurements from the motor sets to the hub periodically; control any chosen motor set with the "hub" device using MQTT messages.

5. Intended audience

People with at least basic knowledge of electronics and wireless communication.

6. Scope

This document includes short descriptions of the external hardware and software used in the project, as well as their relevancy. In addition, it provides the design choices for the project, problems encountered during development, some discarded designs and the results of development. Finally, it discusses the development of the project and provides some recommendations regarding any potential future work for it.

dain0084@edu.eal.dk
juan0314@edu.eal.dk

7. Responsibilities

Name	Responsibility	E-mail
Dainius Čeliauskas	Software & Management	dain0084@edu.eal.dk
Juan Mohamad	Hardware design & assembly	juan0314@edu.eal.dk

8. Repository

This project has a GitHub repository, which contains the code used by the project, the timeline, project plan, electronic schematic files, testing videos, additional media and more: https://github.com/dainezasc/Remote-voice-control-system

II. Background

This chapter describes the hardware components, software (scripts, libraries, etc.), protocols from sources outside the project and their purpose within it.

1. Hardware

1) Raspberry Pi 3 Model B and B+

Figure 1: Raspberry Pi 3 Model B. Sourced from its product page of its official website https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

Figure 2: Raspberry Pi 3 Model B+. Sourced from its product page of its official website https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

The Raspberry Pi 3 Model B and Model B+ are single-board computers developed by the Raspberry Pi Foundation. The Model B was released in 2016, while the Model B+ came out in March 2018 as

a direct upgrade. They are widely used as both educational tools for learning computer science and embedded development boards for hobbyist projects.

Both of the models feature wireless LAN, Bluetooth support and four USB ports for peripherals, as well as GPIO capabilities. The Model B+ has a faster processor (clocked at 1.4GHz versus the Model B's 1.2GHz) and support for faster networks, but is otherwise very similar and can be used interchangeably with the Model B in most cases.

The Raspberry Pi has an ARM CPU, which are common in smartphones, tablets and other embedded systems. Because of this, comparing its performance with x86 CPUs (used in desktop PCs and laptops) with a similar clock speed is difficult.

The GPIO pins allow connecting the Raspberry to microcontrollers, buttons, LEDs, relays and other electronic components, provided that the software to control them is written.

The Raspberry Pi uses SD cards of various sizes for storing an operating system and program memory. It supports various operating systems, with Raspbian being most widely used as it is specially designed for the computer and highly optimized for the Raspberry's hardware.

The Raspberry Pi provides general purpose digital input/output (GPIO) pins that can be used for reading or outputting digital logic signals.

The Raspberry Pi 3 is the main hardware component of the project. Both the Model B and B+ were used. Many of its features were utilized in the project, such as the GPIO for connecting other hardware and wireless LAN support for remote connection, login and MQTT. For the project the Raspbian Stretch Lite operating system was used.

dain0084@edu.eal.dk
juan0314@edu.eal.dk

2) 2WD Robot chassis with motors

Figure 3: The robot chassis model used in the project. Sourced from: https://www.robotshop.com/eu/en/2wd-beginner-robot-chassis.html

This type of plastic robot chassis is widely used in electronics and robotics projects because of its availability and low price. It was chosen to demonstrate the capabilities of the project.

The chassis set includes a couple of 60mm diameter wheels and two DC 120:1 gear motors.
The motor rating for the motors at 5V is 1kg*cm torque and 83 RPM.

3) L293NE H-bridge IC

Figure 4: The L293NE integrated circuit chip. Source from: https://www.adafruit.com/product/807

The L293x series of integrated circuits are H-bridge motor drivers that can provide up to 1A (600mA for the L293D) of bidirectional drive current. It is the cheapest and simplest way to control the state and direction of multiple small DC motors at once. The L293D and the L293NE ICs are functionally the same, except for the L293D having inbuilt protection diodes for inductive spikes.

dain0084@edu.eal.dk
juan0314@edu.eal.dk

The L293x ICs have three pins for each motor it can be connected to: an enable pin and two direction pins for bidirectional control. The state of the pins can be controlled with the Raspberry Pi GPIO, making it easy to change the speed and direction of the connected motors using scripts.

The L293NE IC was chosen for the project since it was easily available and suitable for the DC motors that were chosen. However, it should be noted that the IC can only handle currents up to 1A and voltages up to 36V, meaning that if the motors and the power supply were scaled up, it might need to be replaced with a different motor driver or a custom built H-bridge.

4) TCRT5000 reflective sensor

Figure 5: The TCRT5000 reflective sensor. Source from: http://breadfruit.me/home/291-tcrt5000-reflective-infrared-optical-sensor-photoelectric-switches.html

The TCRT5000(L) are reflective sensors that are composed of an IR diode which emits infrared light and a phototransistor with an inbuilt daylight blocking filter. The arrangement of the IR emitter and the phototransistor allows for passing or blocking current based on how the infrared light is reflected from the surface the sensor is pointed at, which can be used to sense different colors or patterns. The TCRT5000(L) sensors are commonly used in electronics projects for their use along with microcontrollers for detecting or following colored lines with robots.

In the project the TCRT5000 sensor was used as part of a tachometer which could measure the RPM of the wheels and send the value to the Raspberry Pi.

5) LM293 operation comparator

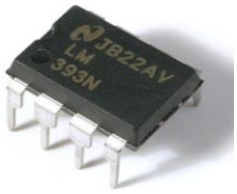


Figure 6: The LM293 integrated circuit chip. Source from: https://tinkbox.ph/store/ic/lm393-low-power-dual-comparator-ic

The LMx93 series of integrated circuits consist of two independent low voltage comparators designed specifically to operate from a single power supply over a wide range of voltages. Operation from dual power supplies is also possible as long as the difference between the two supplies is from 2V to 36V, and VCC is at least 1.5V more positive than the input common-mode

dain0084@edu.eal.dk
juan0314@edu.eal.dk

voltage (sum of the inverting and non-inverting inputs, divided by 2). Current drain is independent of the supply voltage.

In the project one of the comparators of the LM293-N was used as a part of a tachometer to compare the voltages from the sensor and the adjusted potentiometer; based on this information the correct output is sent indicating the sensor readings to the Raspberry Pi.

6) Other hardware components

Other hardware components used in the project:

- Assortment of passive electronic components (capacitors, resistors, connectors, etc.)
- LED for determining on/off states
- 9V battery for powering the motors
- 3.5 inch display for Raspberry Pi - used in the "hub" device to display its IP address
- USB microphone - the Raspberry Pi's has a 3.5mm audio jack, but it cannot be used for audio input - only input from USB works
- (Optional) 5V powerbank for powering the Raspberry Pi

2. External software and protocols

1) Raspbian Stretch Lite

Raspbian is a free operating system developed by Mike Thompson and Peter Green for the Raspberry Pi hardware. The initial build for Raspbian was completed in June 2012, however the operating system is still under active development. Raspbian is based on and is designed to be as close as the Debian operating system as possible, as Debian is popular, well documented and therefore considered high-quality by the Linux community. Because of this, they both share a lot of similarities, to the point where the vast majority of documentation for Debian can apply to Raspbian as well.

Raspbian Stretch is the latest version of the operating system, released on June 2017. The most relevant change from the previous version for the project is ALSA (Advanced Linux Sound Architecture) now being the default sound architecture, with other compatible architectures being available for download. ALSA is responsible for providing audio functionality to the Linux OS. More information about the ALSA project can be found in its homepage (see References).

Raspbian Stretch Lite is a version of Raspbian Stretch that only includes the command line interface (terminal), while leaving out the desktop or any graphical user interface that the original OS includes. This results in vastly decreased size, about 1.4 GB, compared to roughly 4.6 GB of used space by Raspbian Stretch.

The Raspberry Pi that is used for the project uses Raspbian Stretch Lite as its OS since there will be no display (except for the hub device), therefore the desktop will not be required. No keyboards or

mice will be connected to the Raspberry Pi, making it “headless”. Interfacing with the Raspberry will be done by remote connection through SSH using Putty.

SSH is disabled on Raspbian by default. In order to enable it, a blank file named “ssh” or “ssh.txt” must be placed in the “boot” folder of the SD card the OS is installed in, before booting up the Raspberry. After the boot, SSH should be permanently enabled.

More information about Raspbian can be found in its official website (see References).

The Raspberry Pi Foundation also provides images for Raspbian Stretch and its Lite counterpart (see References).

2) RPi.GPIO

RPi.GPIO is a Python package used for controlling the GPIO pins of the Raspberry Pi. It comes with the Raspbian OS.

The package is used in the project to control GPIO behavior: changing pulse-width modulation (PWM) to alter motor speed, detecting falling and rising edges from the output of the tachometer and much more.

Documentation of the RPi.GPIO source code can be found in its Sourceforge page (see References).

3) CMUSphinx (Sphinxbase)

The CMUSphinx toolkit is an open source speech recognition toolkit developed at the Carnegie Mellon University. It encompasses a series of open source speech recognition software, the earliest of which - Sphinx 2 - was developed in 2000. Since then a number of speech recognizers have been developed and maintained by the community (Sphinx 2 - 4, PocketSphinx, acoustic model trainer SphinxTrain). All of these recognizers and the CMUSphinx trainer share common libraries labeled “Sphinxbase”, which must be installed before adding other speech recognizer modules.

Sphinxbase is required for the speech recognition software used in the project to work properly.

The readme file in the GitHub link for Sphinxbase goes into more detail regarding its installation and terms of use (see References).

4) PocketSphinx

PocketSphinx is part of the series of CMUSphinx speech recognition systems. It is a version of CMUSphinx optimized for handheld devices and designed to be lightweight and portable. It is reported to work on mobile, desktop devices and embedded systems (e.g. ARM processor based systems). PocketSphinx is currently considered to be an early release of a research system and is under active development, however, the latest version of PocketSphinx - 5prealpha - has remained

stable for roughly 3 years and already used in other publically available software. Therefore, it was deemed suitable for use in the project.

PocketSphinx was chosen over competing speech recognition software such as Google Voice API for multiple reasons:

- Does not require an internet connection to work - more reliability in remote locations and faster initialization as it does not have to wait for connection to network
- Open source - easy to modify to suit different needs (for example, changing variables to increase speed at the expense of accuracy)
- Lightweight and works with embedded systems - possibility to port the software over to different systems if needed

Since PocketSphinx is open source pre-alpha software, the official documentation for it is somewhat scattered. Most of it can be found in the official CMUSphinx website, including the installation and configuration (see References).

PocketSphinx is compatible with the Raspberry Pi, however, even the newest models (Raspberry Pi 3 Model B or B+) are unable to handle extremely large vocabularies, rendering them unsuitable for tasks such as speech to text or language translation. Fortunately, the task of the project only requires an extremely small vocabulary (about 15-20 words at most), therefore PocketSphinx is expected to perform reasonably fast on the hardware used.

5) Lmtool

Lmtool is a web based tool developed in the Carnegie Mellon University in the early 1990s. It is used for building lexical and language modeling files (together called the decoder knowledge base) for CMUSphinx-based and other speech decoders.

Lmtool is available in this website, along with instructions on how to use it and a FAQ explaining it in more detail:
http://www.speech.cs.cmu.edu/tools/lmtool-new.html

6) Simplified Wrapper and Interface Generator (SWIG)

SWIG is open source software available since 1996 and is in active development. It is meant to connect programs or libraries written in C or C++ with high-level programming languages such as Lua, Python and others.

In the project SWIG is used to allow the Python interface for PocketSphinx (pocketsphinx-python) to connect to the CMUSphinx and PocketSphinx libraries that are written in C.

SWIG must be installed for the Python interface to PocketSphinx to work properly.

More information about SWIG can be found in the official website (see References).

7) Pocketsphinx-python

Pocketsphinx-python is a Python interface to the speech recognition engine PocketSphinx that is used for the project. Even though PocketSphinx has limited Python support on its own, it can be cumbersome to use and is not well supported on some systems. This interface can provide more portability and can be used to easily implement PocketSphinx into standalone Python scripts.

The project utilizes pocketsphinx-python for easier implementation and modification of PocketSphinx in the main Python script. It also allows for easy portability and synchronization of Python scripts and other files between multiple devices.

The readme file in the GitHub link for pocketsphinx-python goes into more detail regarding its installation and terms of use, as well as some example code that will be implemented into the project (see References).

8) Message Queuing Telemetry Transport (MQTT)

MQTT is a machine-to-machine publish-subscribe based messaging protocol first introduced in 1999 by Dr Andy Stanford-Clark of IBM and Arlen Nipper and is now used in numerous applications, including Facebook Messenger. MQTT is designed to be extremely lightweight and efficient at sending small amounts of data through networks reliably and quickly, making the protocol ideal for IoT (Internet of Things) platforms. The devices that communicate through MQTT are called clients and can be divided into three categories based on their purpose:

- Publishers - these devices use MQTT to send short messages or commands with specific topics to the message broker. The topics do not need to be pre-configured - they are created when they are published on, meaning that they can be defined or even randomly generated along with the messages that are sent.
- Subscribers - these devices establish constant or periodic connection with the message broker and monitor a specific topic or set of topics for any new messages. Once the message is received from the broker, it can be displayed or used as a variable in code.
- Brokers - they are responsible for receiving all messages from the publishers and distributing them to certain subscribers based on the topic or set of topics they were published on. The brokers can be based on local devices or websites (public brokers).

Note that a single client can fulfill multiple and even all three of these roles (e.g., when sending an MQTT message to itself).

In the project the MQTT protocol was implemented with Eclipse Paho for transmitting commands and various measurements between devices.

More information about MQTT regarding its other features such as quality of service and encryption can be found in the official website (see References).

dain0084@edu.eal.dk
juan0314@edu.eal.dk

9) Eclipse Paho (paho-mqtt)

Eclipse Paho is an open-source client implementation of the MQTT protocol developed in 2014 and maintained by the Eclipse Foundation and IBM. It is available on a wide variety of platforms and programming languages, including Java, C and Python.

Eclipse Foundation also provides a public MQTT broker for testing devices running Paho, iot.eclipse.org.

The project uses the Eclipse Paho MQTT Python client library for implementing the MQTT protocol and the public broker iot.eclipse.org for communicating between clients.

More information about Paho can be found in the official website (see References).

The description of the source code for the Paho MQTT Python client library can be found in its Python Package Index page (see References).

10) Putty

Putty is a free open source terminal emulator, serial console and network file transfer application, developed by Simon Tatham and initially released in 1999. It is most commonly used to transfer files with the SCP protocol, remotely login to computer systems using SSH and connect to serial ports.

Putty is used in the project to establish a secure connection to the Raspberry Pi and open a terminal for remote login, script execution and debugging.

More information on Putty can be found in the official website (see References).

11) Other external software

- **WinSCP -** used in the project for quickly sharing files between devices using the SCP protocol. Even though Putty already has SCP support, WinSCP features a graphical user interface, an integrated text editor and can work alongside Putty.
- **Advanced IP Scanner** - used to find the IP address of the Raspberry Pi in LAN.

III. Design

This chapter covers the hardware and software design choices for the project along with the various issues found during its development. Moreover, it documents some of the more notable earlier design choices that were chosen and later scrapped.

1. Overview

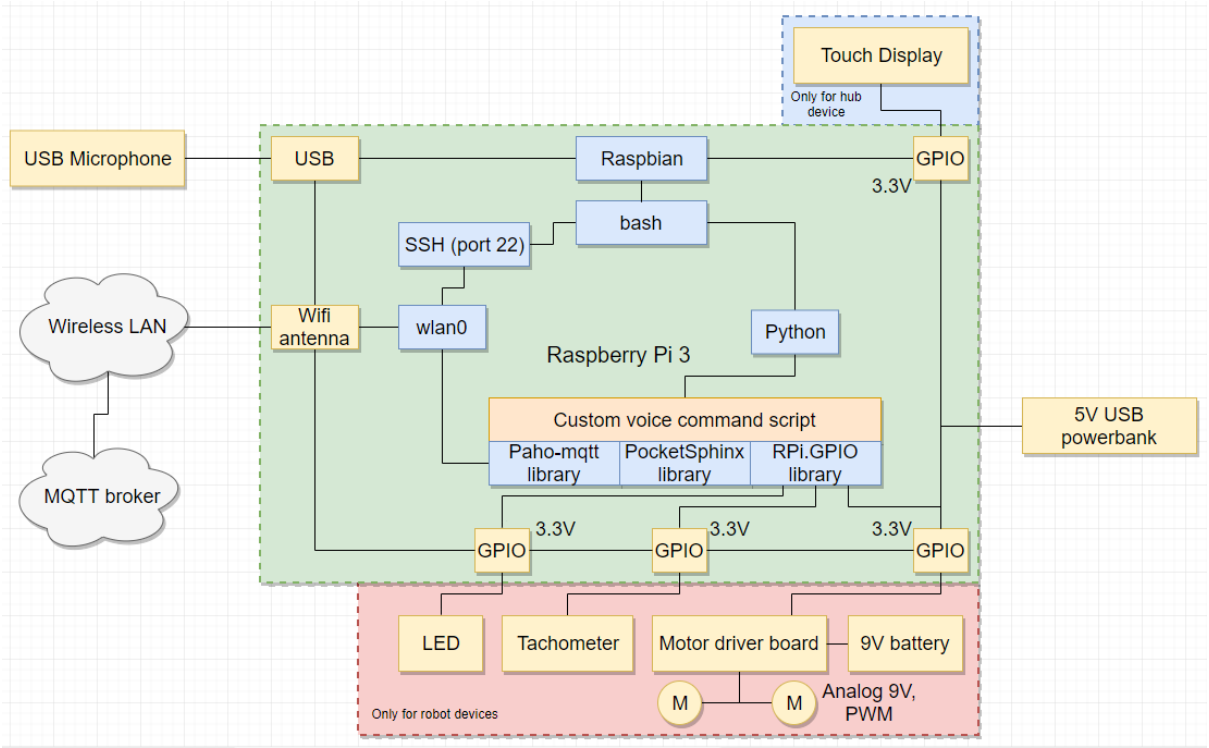


Figure 7: The overview diagram for the system of the project

The Raspberry Pi 3 is the main part of the system, being responsible for connecting all other parts of the system together. It houses the main script and the libraries required to run it, as well as connecting to the online MQTT broker and giving logic signals to connected peripherals.

Interfacing with the Raspberry is done by remote connection using SSH with Putty. This requires the Raspberry to be connected to LAN.

The Raspberry Pi uses the GPIO pins to control the motor driver board and a LED, in addition reading the input from the tachometer module. An external microphone is connected to one of the Raspberry's USB ports for audio input as well. Since there are enough GPIO pins for all hardware modules, no port expanders (I2C or SPI) are needed.

2. Hardware

1) Motor driver board

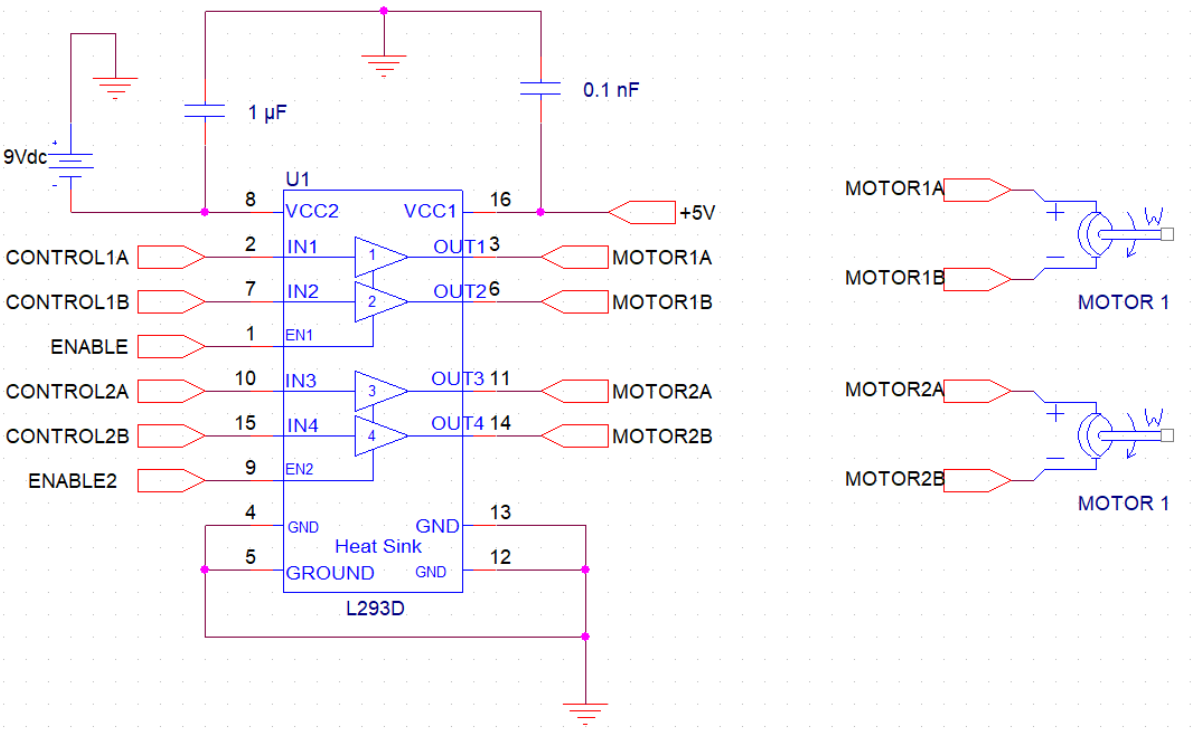


Figure 8: The schematic for the motor driver board circuit

The main component of the circuit is the L293NE motor driver IC, which controls two connected DC motors. It sends DC signal outputs based on the enable and control input pins' state, which is controlled by the Raspberry Pi GPIO. In addition, it is connected to separate 5V (used for logic) and 9V (for powering the DC motors) power supplies, coupled with some bypass capacitors (further described in the Tachometer section).

The Raspberry GPIO is capable of putting out PWM signals to any of its pins. They are used for the enable pins of the motors to change their duty cycle (speed).

The pinout for the L293NE is as follows:

1. Enable/disable pin for the left side motor. It can be used with PWM to change the speed of the motor.
2. Input pin, used to receive logic signals for controlling the left side motor with the Raspberry Pi.
3. Output pin, used to send power to the left motor.
4. Ground.
5. Ground.
6. Output pin, used to send power to the left motor.
7. Input pin, used to receive logic signals for controlling the left side motor with the Raspberry Pi.

dain0084@edu.eal.dk
juan0314@edu.eal.dk

8. Power supply pin for the motors (9V).
9. Enable/disable pin for the right side motor. It can be used with PWM to change the speed of the motor.
10. Input pin, used to receive logic signals for controlling the right side motor with the Raspberry Pi.
11. Output pin, used to send power to the right motor.
12. Ground.
13. Ground.
14. Output pin, used to send power to the right motor.
15. Input pin, used to receive logic signals for controlling the right side motor with the Raspberry Pi.
16. Power supply pin for the IC (max 5V).

In order to be able to drive the motors, the enable pins for each motor need to be driven high. The left motor will drive when either the 2nd or 7th pin is high (not both), while the right motor will drive when either the 10th or 15th pin is high (not both).

2) Tachometer

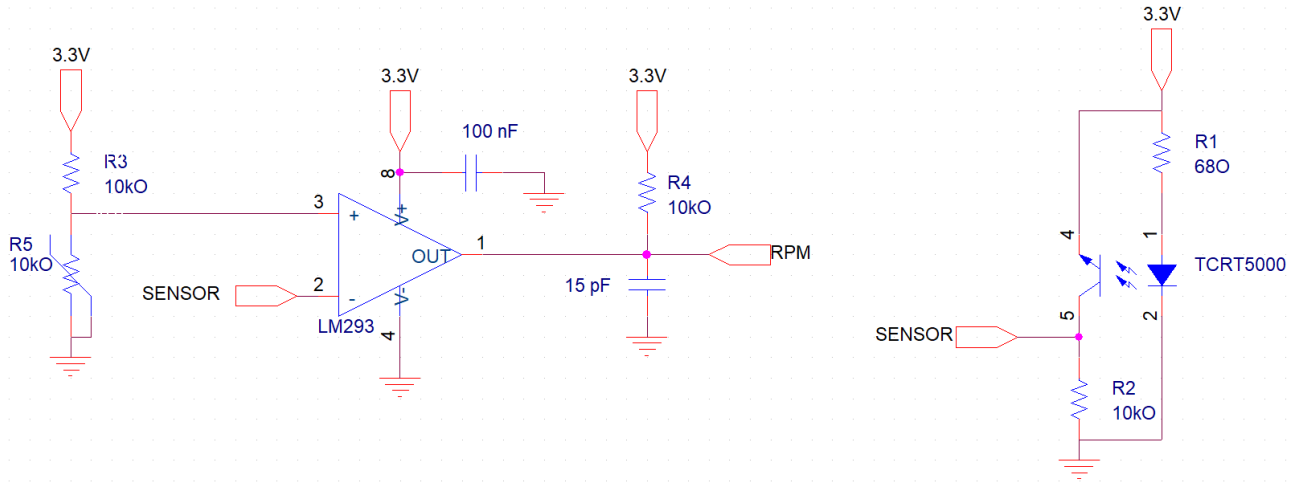


Figure 9: The schematic for the tachometer circuit

The primary component of the circuit is the TCRT5000 reflective sensor consisting of a IR photodiode and a phototransistor. The more infrared light is reflected back to the photodiode, the more current the phototransistor allows to flow through it. When the current flows through the 10k ohm resistor, a potential difference (voltage) is created, and according to Ohm's law (V=I*R) this voltage is directly proportional to the flowing current since the resistance value is constant. Therefore the more infrared light is reflected back, the more the voltage at the 10k resistor increases.

This voltage is compared to the constant reference voltage value created with the potentiometer using one of the LM293 comparators. The potentiometer is connected to the non-inverting input and the sensor is connected to the inverting input. Whenever the voltage at the non-inverting input is

dain0084@edu.eal.dk
juan0314@edu.eal.dk

bigger than the inverting input, the output from the comparator is 3.3V, or HIGH. The potentiometer is adjusted in such a way that when there is very little or no infrared light reflecting back to the sensor, the reference (non-inverting) voltage is bigger than the sensor (inverting) voltage.

As a result, when the sensor is pointed at a black object, the output from the comparator will be LOW, while if is pointed at a white object, the output will become HIGH. The sensor is pointed towards one of the tires of the robot, which are part of the wheels connected to the motors. The tire is entirely colored black except for a small white strip. The strip is used by the tachometer as an indicator for when the wheel has finished a rotation.

The pin marked RPM sends the output from the comparator to the Raspberry Pi GPIO. This output signal is used to measure the RPM of the wheel; the RPM value is constantly incremented by each rotation and stored for later use.

Theoretically, the RPM value is incremented when a falling edge is detected by the Raspberry GPIO. However, in practice the falling edge detection rate will be far more frequent than the actual wheel rotation frequency because of switch bounce:

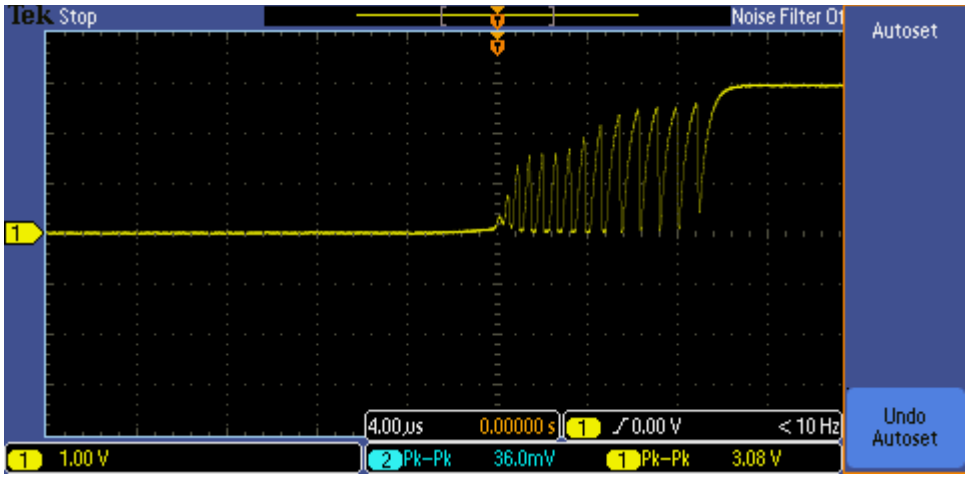


Figure 10: Unwanted switch bounce resulting from AC noise when changing reflections slowly

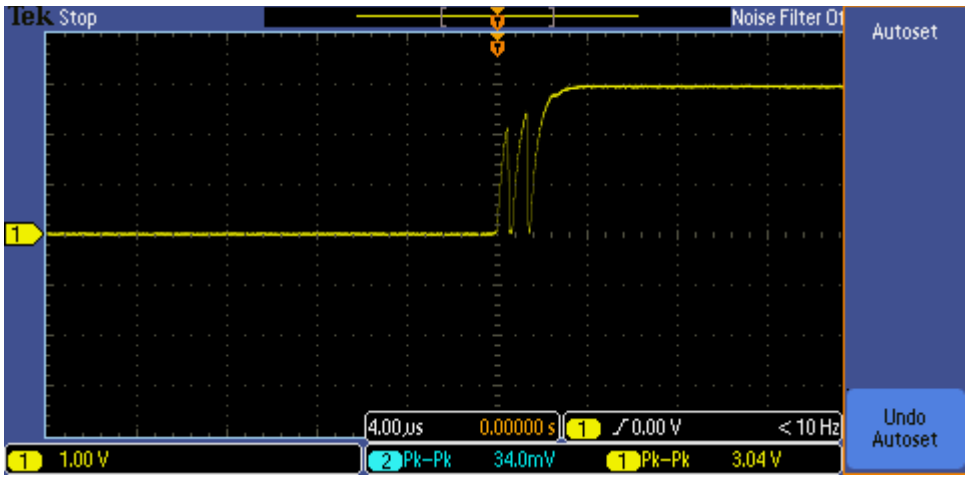


Figure 11: Unwanted switch bounce resulting from AC noise when changing reflections fast

dain0084@edu.eal.dk
juan0314@edu.eal.dk

The switch bounce results in high frequency AC noise, which causes voltage spikes and, therefore, lots of falling edges. Because of this the RPM is incremented at a much faster rate than it is actually supposed to, presenting a major problem.

The problem was solved by introducing software debouncing and bypass capacitors between the VCC and ground. The bypass capacitors are designed to act as short circuits to ground for AC noise, while acting as open circuits for DC signals. This way any AC noise that is present on a DC signal is removed, producing a much cleaner and pure DC signal.

3) Connection to Raspberry Pi GPIO

These are the schematics showing how all hardware modules connect to the Raspberry GPIO:

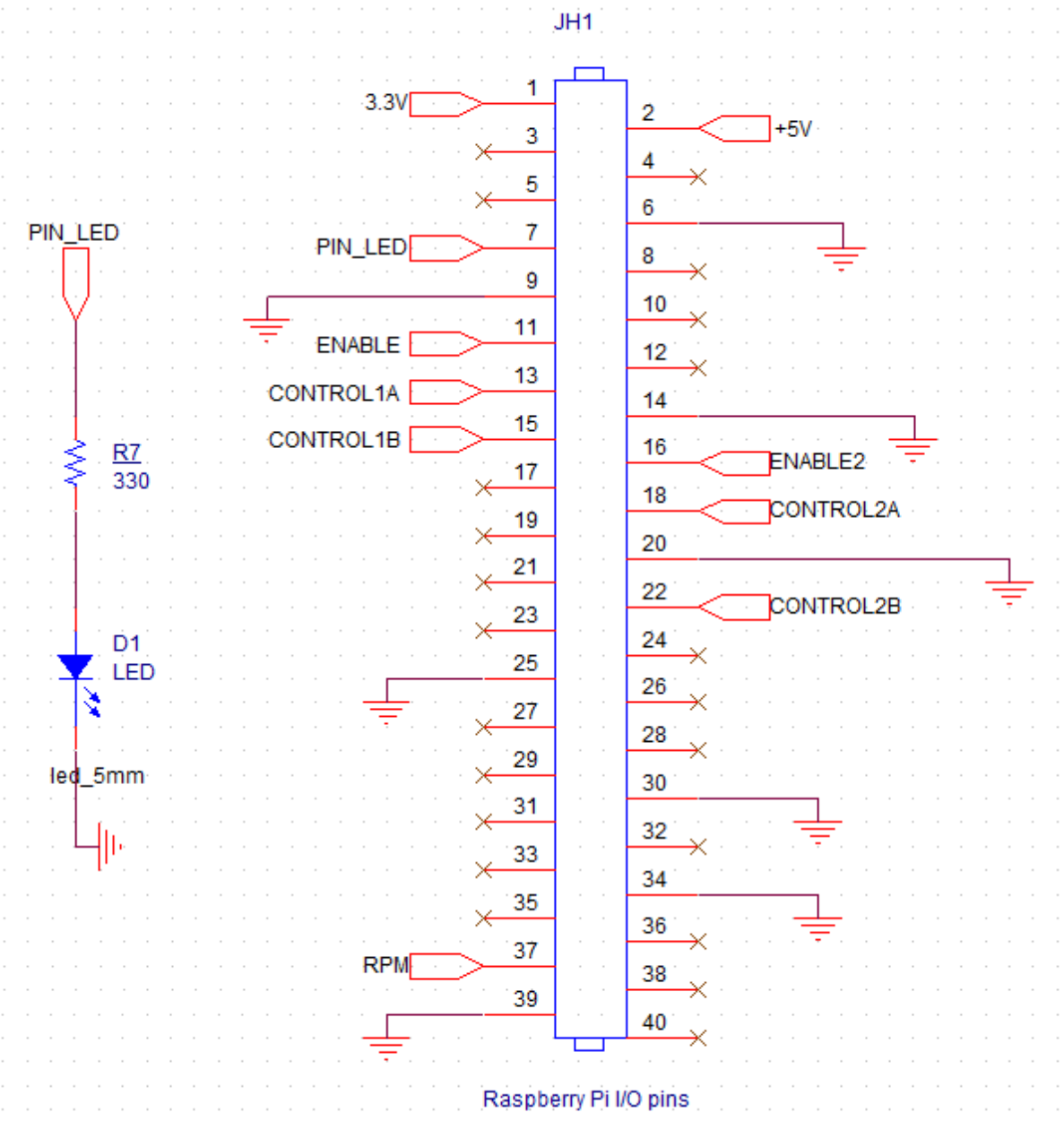


Figure 12: Schematic of the Raspberry GPIO and all connections to it, with LED board

dain0084@edu.eal.dk
juan0314@edu.eal.dk

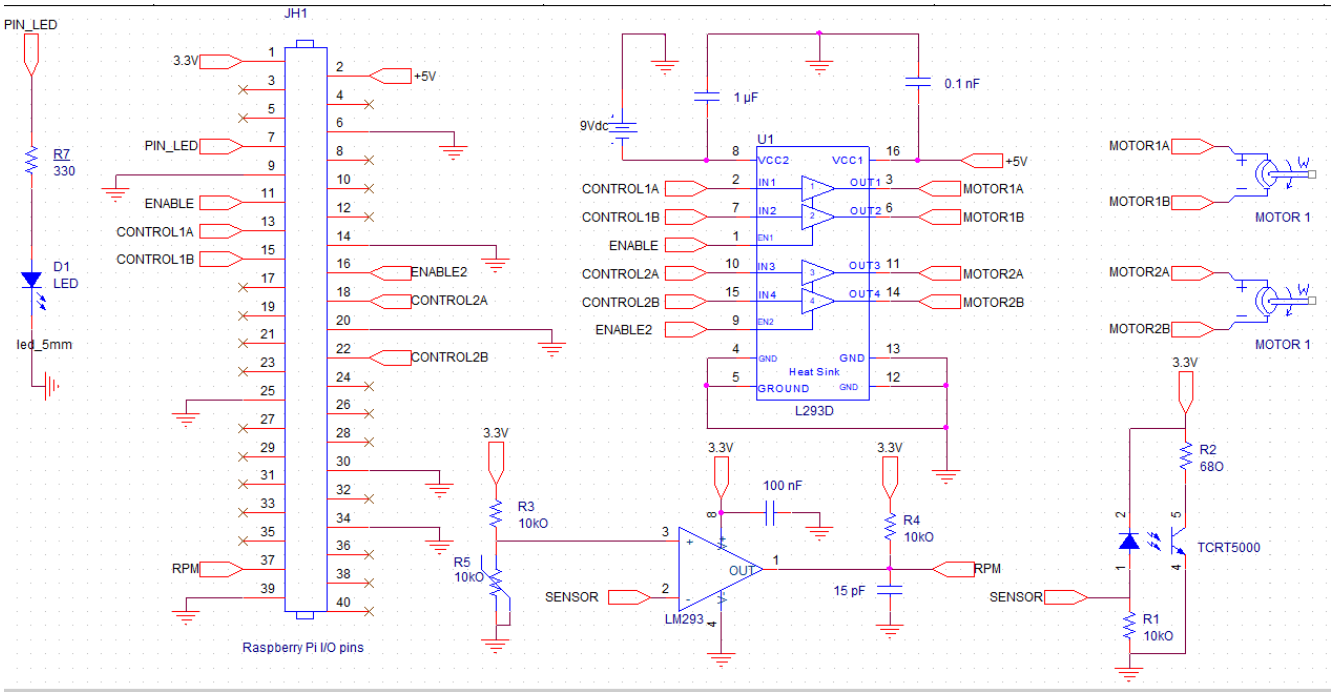


Figure 13: Complete schematic of all circuits in the project, with connections between them shown

In addition a LED has been connected to one of the pins of Raspberry GPIO for use in the software.

3. Software

1) Wireless configuration (wpa_supplicant)

In order to make a headless Raspberry Pi automatically connect to a specific network, the network's properties must be specified in the "wpa_supplicant.conf" file. It usually needs to be created and put in the required directory manually, but if it is already created, its contents should look similar to this:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=GB

network={
        ssid="[Wifi network name]"
        psk="[Wifi network password]"
}
```

If the correct network ID and password are put into the `ssid` and `psk` fields, the Raspberry will automatically connect to this network after booting.

It is possible to assign multiple networks and prioritize them in the file:

```
network={
        ssid="[Wifi network name]"
        psk="[Wifi network password]"
```


dain0084@edu.eal.dk
juan0314@edu.eal.dk

```
        priority=1
}
network={
        ssid="[Wifi network name]"
        psk="[Wifi network password]"
        priority=2
}
...
```

If the first priority network is unreachable, the Raspberry will attempt to connect to the second/third/etc. priority network.

This setup works for the vast majority of networks, assuming the ID and password is known. However, during the development of the project it was often required to connect the Raspberry to a PEAP protected network. The PEAPv0/EAP-MSCHAPv2 authentication protocol checks for an unique ID and a password associated with the ID to grant access to the network. In a desktop PC or laptop the authentication process would be simple - all that is needed is to type the ID and the password. Connecting the Raspberry to the network is slightly more tricky:

```
network={
      ssid="[Wifi network name]"
      proto=RSN
      key_mgmt=WPA-EAP
      pairwise=CCMP
      auth_alg=OPEN
      eap=PEAP
      identity="[Wifi unique identity]"
      password="[Wifi unique password]"
      phase1="peaplabel=0"
      phase2="auth=MSCHAPV2"
}
```

In this case the `password` field does not contain the real password, but an MD4 hash string generated from the real password. It can be obtain with the command:

```
echo -n 'YOUR_REAL_PASSWORD' | iconv -t utf16le | openssl md4 > hash.txt
```

A new file “hash.txt” will be created, with its contents being similar to this:

```
(stdin)= a6fsfs71xxxxxxxxxxxxxxxetcetcetc
```

Therefore the `password` field should look like this:

```
password=hash:a6fsfs71xxxxxxxxxxxxxxxetcetcetc
```

Now the Raspberry should be able to connect to the network.

The “wpa_supplicant.conf” file is located in the /etc/wpa_supplicant/ directory, which cannot be accessed with a machine running Windows. However, if a “wpa_supplicant.conf” file is found in

the “boot” folder inside the SD card, it is automatically moved to the correct directory next time the Raspberry is booted, same as with the “ssh” file.

2) Initial setup

Once the Raspberry has SSH enabled, is connected into the network and its IP address is known, its command line terminal can now be accessed with Putty.

Before any code that uses PocketSphinx is written, the speech recognizer should be tested first. In order to test it, the audio input needs to be correctly configured; to be more precise, the USB microphone needs to be set as the default audio input device for the Raspberry. The best way to do this is by editing some ALSA configuration files. The alsa.conf file in the /usr/share/alsa/ directory contains these lines:

```
defaults.ctl.card 0
defaults.pcm.card 0
```

Which should be changed to:

```
defaults.ctl.card 1
defaults.pcm.card 1
```

In addition a file named “.asoundrc” should be created in the /home/pi/ directory with the following contents:

```
pcm.!default {
 type hw
 card 1
}

ctl.!default {
 type hw
 card 1
}
```

This should set the microphone as the default audio input device. The audio input can be further configured (increased volume, etc.) using the `alsamixer` command.

The audio input can be tested with the `arecord -D plughw:1,0 test.wav` command; it will continuously record audio (unless stopped with Ctrl-C) and create a .wav file, which can be exported or played from the Raspberry if an audio output device is connected.

When Sphinxbase and PocketSphinx are installed, the speech recognition engine can be tested by running the pocketsphinx_continuous.c file in the /home/pi/pocketsphinx-5prealpha/src/programs/ directory:

```
./pocketsphinx_continuous -adcdev sysdefault -samprate 48000 -nfft 2048 -inmic yes
```

3) Keyphrase files (keyphrase.dic and keyphrase.list)

The PocketSphinx speech recognition engine supports multiple methods of "searching" for certain words in the dictionary. The default method is recognition with a language model. The language model is a file which contains every word that the program can recognize, and it will attempt to recognize every recorded sound as one of those words, even if it sounds nothing like natural speech. This method is meant to be used for building software which needs to function continuously and use an extensive vocabulary, such as translation or personal assistant programs.

Another method supported by the engine is keyword spotting. In this mode the program looks for specific keyphrases in recorded speech and ignores other sounds. In addition, it allows to configure a "detection threshold" for each keyword - in other words, the speech recognizer can be set to be more sensitive to certain keyphrases (easier to detect), while being less sensitive to others (harder to detect, may require better pronunciation and clearness). This method works best with applications that need smaller vocabularies and more precision. However, it is important to set and calibrate the detection threshold values correctly to prevent both false alarms and missed detections.

Both methods need to have a certain set of files to function (a language model and dictionary file), all of which can be generated using the "lmtool".

For the project's needs the keyword spotting method is the most appropriate since it uses a very small vocabulary (about 15-20 words), ignores background noise and makes it easy to adjust sensitivity to certain more important keywords. These keywords should be able to control:

- Direction of the motors (forward, backward, left, right, stop)
- Speed or duty cycle of the motors
- (Optional) Blinking an inbuilt LED or printing messages on the terminal

The "lmtool" can generate a dictionary file for the project's speech recognition script if a sentence *corpus* file is uploaded. The file consists of all keywords and keyphrases to be used, separated with new lines, arranged in a column and saved in .txt format.
This was the project's sentence corpus file which contains all the keywords that it uses:

```
Begin
Drive
Back
Left
Right
Stop
Blink diode
First
Second
Third
```


dain0084@edu.eal.dk
juan0314@edu.eal.dk

```
Fourth
```

There are a few reasons why the keyword list was designed in such a way.

The keyword “Begin” is meant to be used as a “control” word. In the initial state of the script all other keywords will be recognized, but they will not change the state of the motors unless the keyword “Begin” is uttered first. Upon accepting this keyword all other keywords will execute their functions correctly until the keyword “Stop” is uttered, at which point the script will go back to its initial state.
This keyword was created because the current speech recognition system can often be unreliable (especially in noisy areas) and interpret background noise as random keywords. A system using a “control” word is more reliable since it can only change the state of the motors when explicitly allowed, while ignoring other keywords in its initial state.

The keyword for driving the motors forward is “Drive” because it is easy to spell and relatively distinct from the other keywords, which helps avoid misinterpretation. (in rare cases, the speech recognizer can mistakenly interpret the keyword “Drive” as “Right” and vice versa)

Previously the keyword for driving motors forward was “Forward”. However, it proved to be difficult for PocketSphinx to recognize it when it was spoken with different accents, therefore it was changed to a more easily pronounceable keyword. This also happened with the previous keyword for driving motors backward, “Backward”, which was changed to simply “Back”.

The keyphrase “Blink diode” is used to blink an LED mounted on one of the Raspberry Pi GPIO pins a couple of times. It does not need to be enabled with the keyword “Begin” first, as it does not cause any motor to move and is often used to test if the script is working in its initial state.
This keyphrase shows that the speech recognition script can execute commands other than controlling motors.

The keyword “Stop” is used to stop all currently running motors and prevent other keywords from starting them again unless the keyword “Begin” is spoken.

The keywords “Left” and “Right” cause only one of the motors (the right and left motor, respectively) to operate, causing the whole set to turn left or right.

The keywords “First”, “Second”, “Third” and “Fourth” change the duty cycles of the motors to 25, 50, 75 and 100 percent respectively, similar to switching gears on a manual automobile. In the script’s initial state the duty cycle is set to 25%, or the “first gear”.

When the sentence corpus file is uploaded and compiled by the lmtool, it generates a number of files, however only the dictionary file (.dic) is required. This is the project’s dictionary file “keyphrase.dic”:

```
BLINK B L IH NG K
BEGIN B IH G IH N
```

```
DIODE D AY OW D
BACK  B AE K
DRIVE D R AY V
FIRST F ER S T
FOURTH      F AO R TH
LEFT  L EH F T
RIGHT R AY T
SECOND      S EH K AH N D
STOP  S T AA P
THIRD TH ER D
```

It contains information for the speech recognition engine on how the keywords are pronounced. It could be used to alter the required pronunciation for each keyword.

There is one additional file needed for the speech recognizer to operate in keyword spotting mode - the detection threshold file. It must be created manually, but its contents are easy to understand and edit. The project's keyword detection threshold file "keyphrase.list" looks like this:

```
BLINK DIODE /1e-5/
BEGIN /1e-3/
DRIVE /1e-3/
BACK /1e-4/
LEFT /1e-5/
RIGHT /1e-4/
STOP /1e-15/
FIRST /1e-10/
SECOND /1e-7/
THIRD /1e-7/
FOURTH /1e-7/
```

The values between slashes next to each keyword show their respective thresholds. To better understand these numbers, an example value of 1e-15 will make the speech recognizer extremely sensitive towards the keyword and cause it to be a lot more reliably interpreted, while a value of 1e-1 will vastly decrease sensitivity for the keyword and require the speaker to be very clear and precise when uttering it.
To maximize reliability and prevent both missed detections and false positives, the detection thresholds for each word must be carefully balanced based on its length, pronunciation difficulty and importance. For example, the keywords "Stop" and "First" have the lowest detection thresholds since they are the most important commands for safety of the project - being able to stop or slow down the motors reliably with voice commands is imperative.

Both the dictionary and keyword detection files, along with the language model file that comes with PocketSphinx are required for the main script to work properly.

4) PocketScript (for Hub and Robots)

Both the hub and the motor set Raspberry Pis have the same software setup except for the main script, which is slightly different for each device.

This flowchart illustrates the code of the script for the hub device (see the full script in appendix - 2):

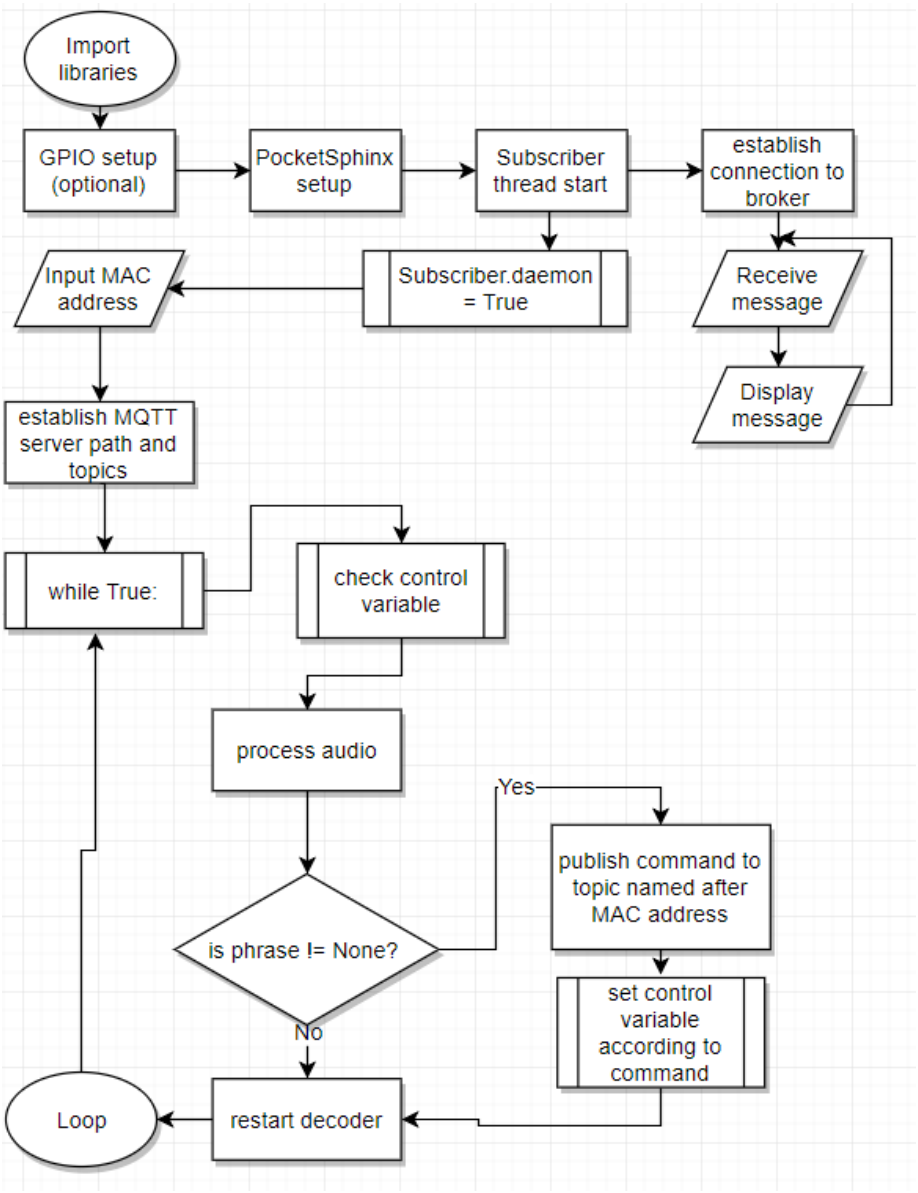


Figure 14: Software flowchart for the hub script

After the speech recognition engine is set up, a connection to the MQTT broker is established on a separate thread, meaning that this function and the rest of the script are running (almost) simultaneously. This is required as the continuous connection to the broker is a blocking function and would not allow the rest of the code to execute if it was included in the main thread. The hub device subscribes to the `voice_mqtt/measurements/#` topic, which includes all sub-topics of each robot device taking their RPM measurements. This way the hub device periodically receives the published RPM values of all active robots. The subscriber thread is then set to be a daemon – if the main thread is stopped, this thread will automatically stop as well.

The script then prints out a message asking to input the MAC address of the robot device for remote control. If all that is needed from the hub is receiving RPM measurements, typing the MAC address is not necessary to keep the script running. However, if remote voice control of any robot is needed, the MAC address should be entered in this format:

```
b8-27-eb-xx-xx-xx
```

When the address is entered, the hub will set the MQTT message publishing topic as voice_mqtt/commands/[input MAC address]. Soon after, the speech decoder is initialized, the microphone begins recording and the loop including the remaining part of the script is entered.

In the start of the loop the Boolean "control" variable that is set by the control words "Begin" and "Stop" is checked. This variable plays a much bigger role in the script for the robots, though in the script for the hub it can inform whether the remotely controlled robot is ready to accept driving commands.

The speech decoder processes and analyzes the audio input from the microphone, listening for the keywords defined in the keyphrase files. When a keyword is detected, it is displayed on the terminal and is subsequently sent as a MQTT message to the remote-controlled robot. The hub does not control the robot directly; the results of the sent message are dependent on the robot's state. For example, if the "control" variable of the robot is False when the message is received, the robot will signal that the control word must be said first, just as it would if the same voice command was attempted using the local microphone. However, the "control" variable of the robot can also be changed from the hub if needed.

Once the command execution is done, the decoder is reset, and the loop begins again.

The script for the robots is similar to the hub, with a few major differences (for the full script see appendix - 3):

dain0084@edu.eal.dk
juan0314@edu.eal.dk

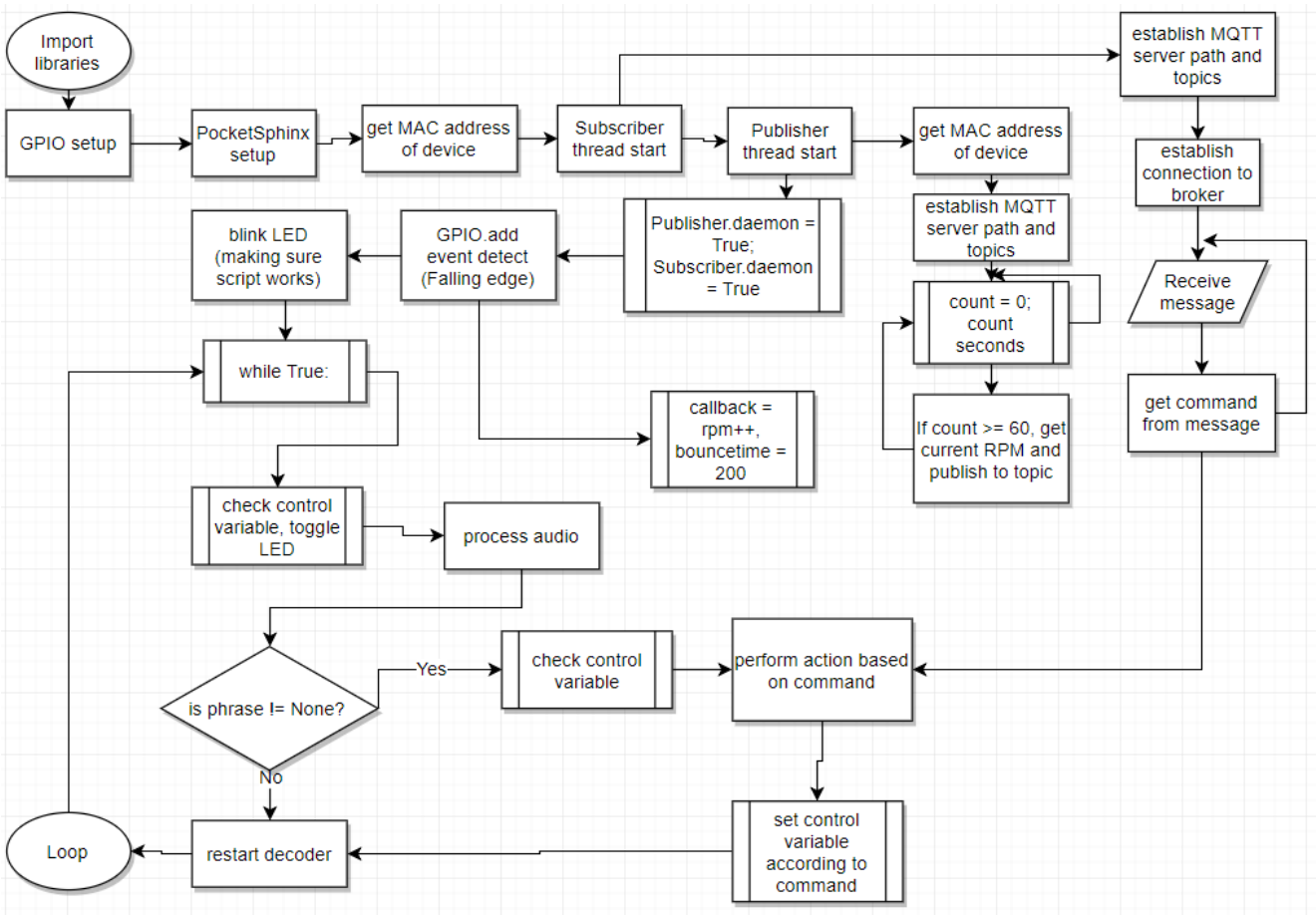


Figure 15: Software flowchart for the robot script

In addition to the speech recognizer, the GPIO that controls the motor driver board and gets the input from the tachometer is also set up. The MAC address of the Raspberry is retrieved and stored.

The most visible change from the script for the hub is the new "publisher" thread, which is responsible for gathering and resetting the current RPM value, as well as sending it to the `voice_mqtt/measurements/[robot's MAC address]` topic as a MQTT message every 60 seconds.

The subscriber thread in the script for the robots is responsible for establishing connection to the MQTT broker and subscribing to the `voice_mqtt/commands/[robot's MAC address]` topic. When a message with this topic is received, the voice command execution function is launched with the message payload (keyword from the hub) as the keyphrase parameter.

After enabling and setting up the "publisher" and "subscriber" threads, the GPIO falling edge detection is launched for the pin used by the tachometer. Whenever a falling edge signaling the sensor detecting the white strip of the tire is detected, a callback function is launched on a separate thread. Its only purpose is to increment the RPM value by 1 every time it is executed.

dain0084@edu.eal.dk
juan0314@edu.eal.dk

Before the loop starts, the LED connected to the Raspberry GPIO is blinked to inform that the script is working and the speech decoder is ready.

In the start of the loop the "control" variable that is set by the control words "Begin" and "Stop" is checked. If the variable is True, the LED connected to the Raspberry GPIO is toggled on, signaling that the motors of the robot can now be controlled with voice commands. The rest of the loop works the same as in the script for the hub, with the exception of the keywords being used for the voice command execution function instead of being passed over through MQTT.

The voice command execution function can blink the LED and control the Raspberry GPIO output to the motor driver board's enable and direction pins, if the control variable is set to True using the "Begin" keyword.

4. Problems and solutions

Here are some of the various problems related to the project that were found in its development, and how some of them were solved.

Problem: How to find the IP address of the Raspberry Pi in the network?
Solution: The easiest way to find all IP addresses in the network using Windows is to run the arp -a command in the Command Prompt. However, it only shows the IP addresses of devices the computer recognizes and/or has interacted with before, which can often result in an incomplete ARP list. Using a dedicated IP scanning program such as Advanced IP Scanner solves this issue reliably, although it is slower than the arp -a command.
In addition, the Raspberry Pi prints its IP address on the terminal upon successful connection to a network as default behavior. If a display is connected to the Raspberry and properly configured, it can be shown from the Raspberry itself when it connects, making it the fastest available solution.

Problem: the speech recognition engine is unreliable (lots of missed detections and false positives) and is messing up the motor set controls.
Solution: this problem was never completely solved during development of the project, but there were a number of different ways discovered for mitigating it. The introduction of the "control" word increased the reliability of the speech recognition script significantly since background noise was no longer capable of randomly changing the behavior of the motors, however this only negates a consequence of this problem, not the source.
The speech recognition script would undoubtedly work more reliably when using better quality microphones and/or noise cancellation techniques - better audio quality would allow for increased keyword detection thresholds, which would result in less false positives.
More time spent on calibration of the keyword thresholds would also help increase reliability.

Problem: the speech recognition script only runs when it is manually executed in the terminal. It needs to run when the Raspberry Pi is booted. How to do that?

Solution: there are a lot of ways to run scripts on the Raspberry Pi when it is booted. One solution is to edit the .bashrc file in the */home/pi/.bashrc* directory by putting this line in the end of the file's contents:

```
sudo python /home/pi/yourprogramhere.py
```

As a bonus, this program will also be executed every time the Raspberry is remotely logged in with SSH, making it useful for debugging. It should be noted that to execute the .bashrc file on boot the "Console Autologin" option must be enabled in *raspi-config* under "Boot options".

Problem: when the script starts on boot, there's no way to immediately tell if it is working.
Solution: a LED was connected to one of the pins of the Raspberry GPIO; upon starting the script the LED would blink a couple of times. In addition, the LED is toggled on after the "control" word is spoken, making it easy to identify if the motor set is ready to be driven.

Problem: what if the microphone stops working while the motor set is driving somewhere? The speech recognition engine will become unresponsive.
Solution: this problem was anticipated from the project's first iteration and has not been fully solved during its development. The current version of the main script cannot be executed without having any form of audio input, however, if it is already running when the input device stops functioning, it will continue its previous command while becoming completely unresponsive (unless using Ctrl-C, but KeyboardInterrupt is not intended to be used).
This issue has been partially solved when support for remote MQTT commands from the hub was introduced. Even if the motor controlling Raspberry's local audio input stops working, it will still listen for remote commands from the hub, making it possible to stop or move the controlled device away from dangerous situations using MQTT.

Problem: The tachometer has a lot of switch bouncing (spikes of voltage) on the rising/falling edges, resulting in wrong RPM recordings.
Solution: The most common way of mitigating switch bounce in electronics is the addition of bypass capacitors between the VCC (power supply) and ground pins; the capacitors short AC signals to ground in a way that any AC noise that is present on a DC signal is removed, producing a much cleaner and pure DC signal.
This issue can also be mitigated in software by "waiting out" the switch bounce after falling/rising edge detection. This can be done using the Raspberry GPIO library using the *bouncetime* parameter in its callback function:

```
GPIO.add_event_detect(channel, GPIO.RISING, callback=my_callback,
bouncetime=200)
```
Or
```
GPIO.add_event_callback(channel, my_callback, bouncetime=200)
```

Bouncetime is the waiting period between an edge detection event and its callback function execution (response to edge detection); it is measured in milliseconds.
These two methods can be combined for even more effective debouncing.

dain0084@edu.eal.dk
juan0314@edu.eal.dk

5. Progress

This chapter presents some of the project's earlier iterations: older speech recognition setups, versions of scripts and network configurations. They were scrapped for various reasons, though some of the lessons learned during their development could prove useful in future works.

1) VoiceMacro

The first version of the project that was built did not have the speech recognition software installed in the Raspberry Pi. Speech processing was done on a PC using an application named VoiceMacro: https://www.voicemacro.net/

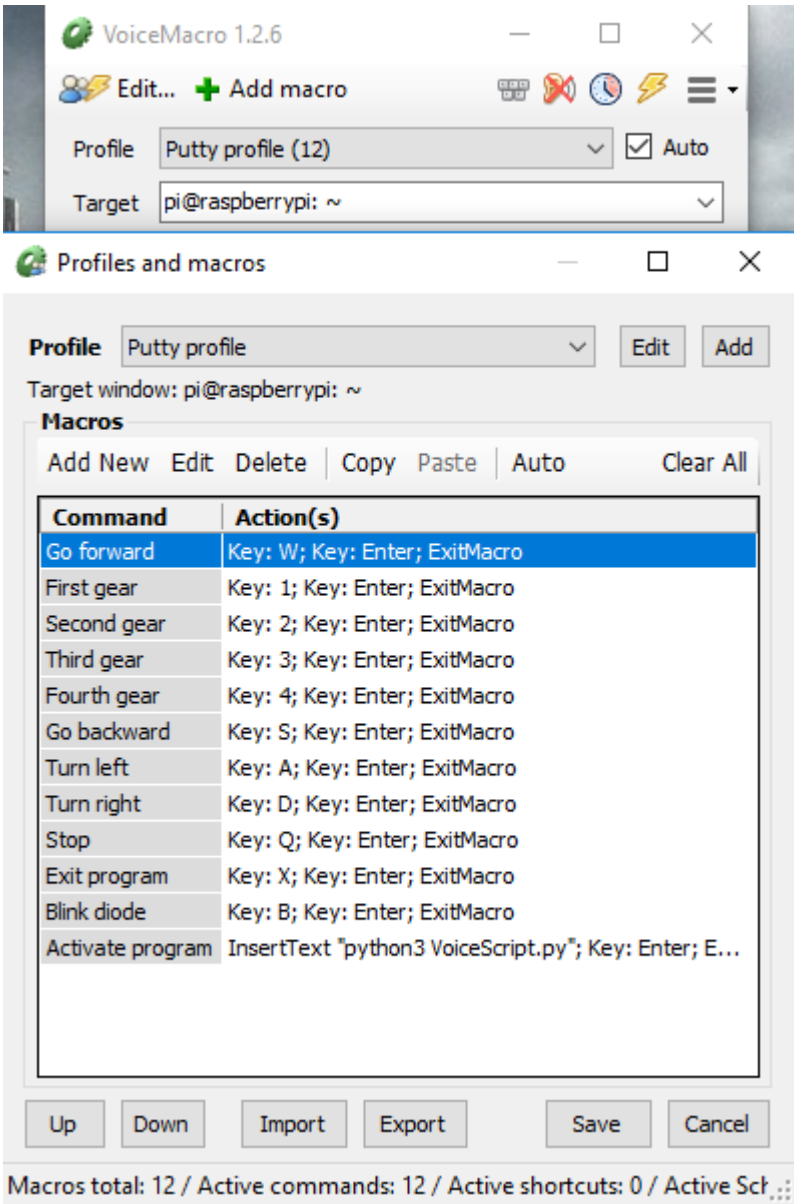


Figure 16: VoiceMacro graphical user interface, with all voice commands used at the time shown

VoiceMacro is used to create macros - custom keyboard shortcuts for performing one or more tasks automatically, such as opening multiple task windows or simulating keyboard presses. Individual "profiles" containing voice commands for specific target windows can also be created, with the option of automatically selecting them before command execution.

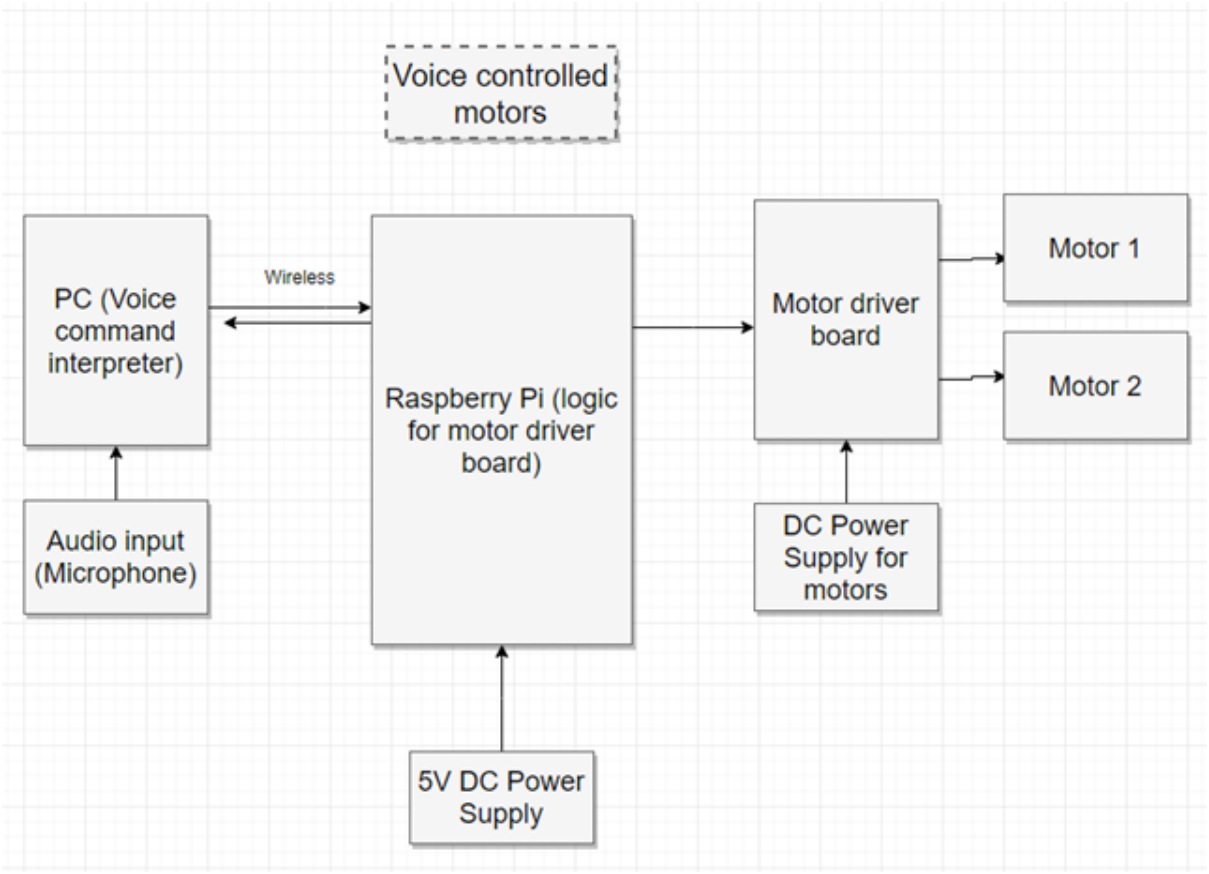


Figure 17: Overview diagram for the system using VoiceMacro

The PC with the microphone was remotely connected to the Raspberry through SSH. The script that was run by the Raspberry (see appendix - 4) executed commands based on keyboard input from the PC, which was simulated using VoiceMacro. This setup, at least in practice, allowed to control the motors with reasonable precision and speed.

While this design was sufficient for completing some of the objectives of the project, it had major issues:

- The necessity of having a PC running Windows OS as part of the project's system since VoiceMacro uses Microsoft .NET Framework and Microsoft Speech API
- The PC and the Raspberry always needing to be on the same LAN, severely decreasing portability

After the idea to move the speech recognition software to the Raspberry itself was formed, this setup was discarded in favor of the current design. However, it did help kickstart the project's development as it was presentable and rather quick to create.

dain0084@edu.eal.dk
juan0314@edu.eal.dk

2) Static IP

One constant annoyance during the project's development was having to rediscover the IP addresses of the Raspberry Pi devices in the network after they were left unconnected for a longer period of time (at least about 2-3 days). One of the proposals to solve this problem was to set static IP addresses for the devices, removing the need to scan the network for them entirely as they would always remain the same.

The method that was chosen to set a static IP address was to edit the `dhcpcd.conf` file in the `/etc/` directory of the Raspberry's SD card by changing these lines:

```
interface eth0
static ip_address=[static IP address for ethernet]
static routers=[default gateway/router IP]
static domain_name_servers=[default gateway/router IP]

interface wlan0
static ip_address=[static IP address for wireless]
static routers=[default gateway/router IP]
static domain_name_servers=[default gateway/router IP]
```

After setting up this and the wpa_supplicant file, the Raspberry should connect to LAN with the defined static IP address.

This setup would theoretically solve the problem of constantly changing IP addresses. However, in practice there were some issues:

- The `/etc/` directory is unreachable when using Windows, prompting the need to use external applications or other similar methods to access it. This makes changing the desired static IP addresses tedious
- In a large, busy network, connection using static IP is very unreliable. Very often the Raspberry was unable to connect to the network at all, possibly because that IP address was already taken by another device in the network at that moment.

It was decided that spending some more time setting up the Raspberries for development was preferable over spotty connections, therefore the choice to set static IP addresses for them was discarded. However, it would most likely prove to be an excellent setup for localized systems in small networks.

3) Local MQTT broker

When MQTT support was first introduced to the project, the hub device was set up to be the local MQTT message broker of the system.

To setup the hub as a local broker, the `MQTT_SERVER` variable should be set to `localhost` in the script for the hub and the hub's IP address in LAN in the script for the robots.

This setup can help localize the system and avoid using public/online MQTT brokers, but it has issues. The script for the robots requires a preset IP address of the hub, which is difficult to determine without periodic configurations if the hub's IP is dynamic and constantly changing. In addition, the hub would always have to be in the same network as the robots to remotely control them and receive measurements, making the hub device mostly pointless.

This design was scrapped when public MQTT brokers were discovered and integrated into the project's system, though it could be useful for localization and avoiding using public/online MQTT brokers.

IV. Results

This chapter presents the results achieved at the end of the project's development, some of the tests that were taken, discussion of the project and its development and some recommendations for future work, should this project's development be continued.

1. Finished prototypes

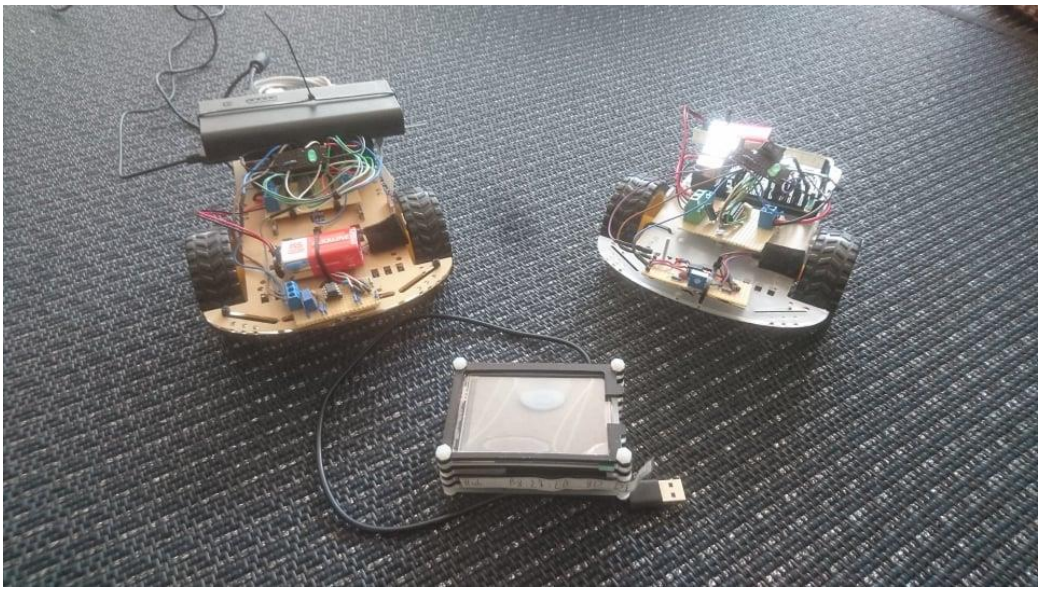


Figure 18: All devices of the project: the hub device and two robot cars

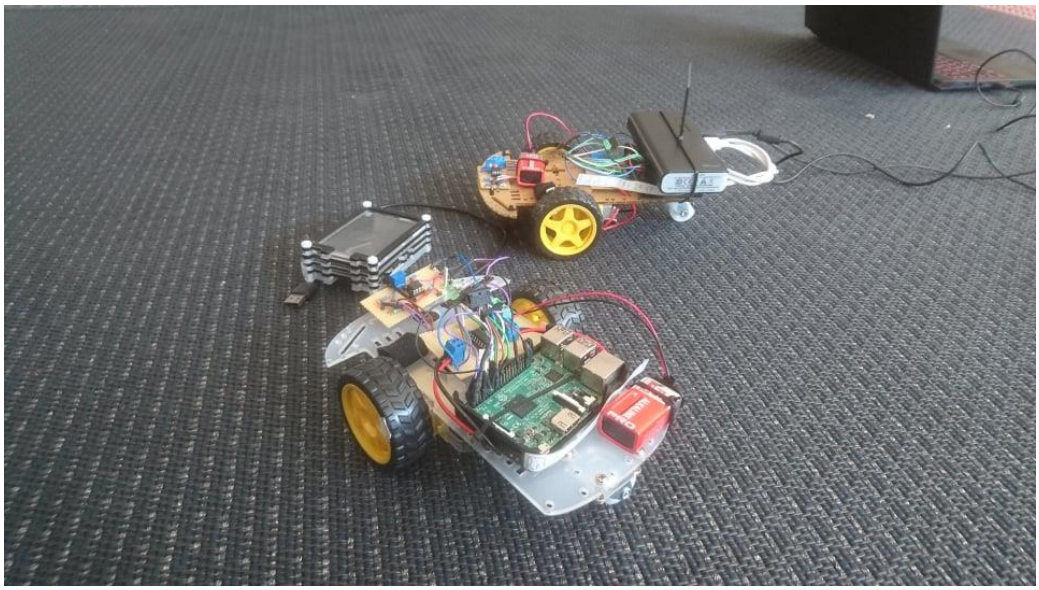


Figure 19: All devices of the project, viewed from right side

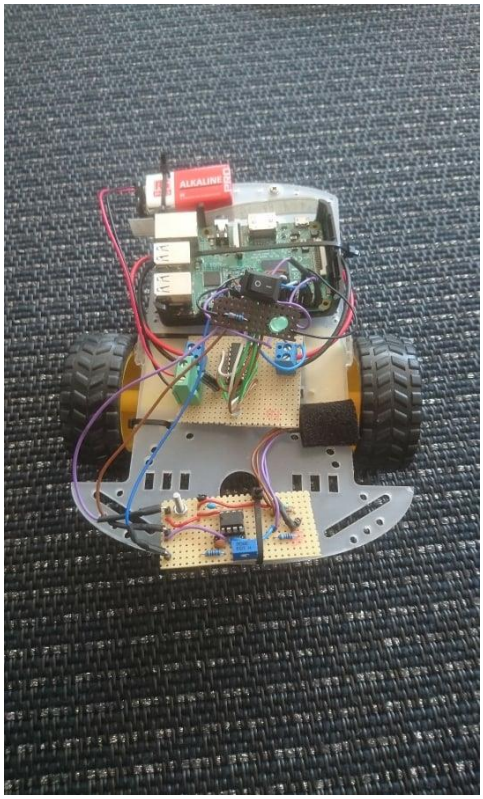


Figure 20: Close-up of one of the robots

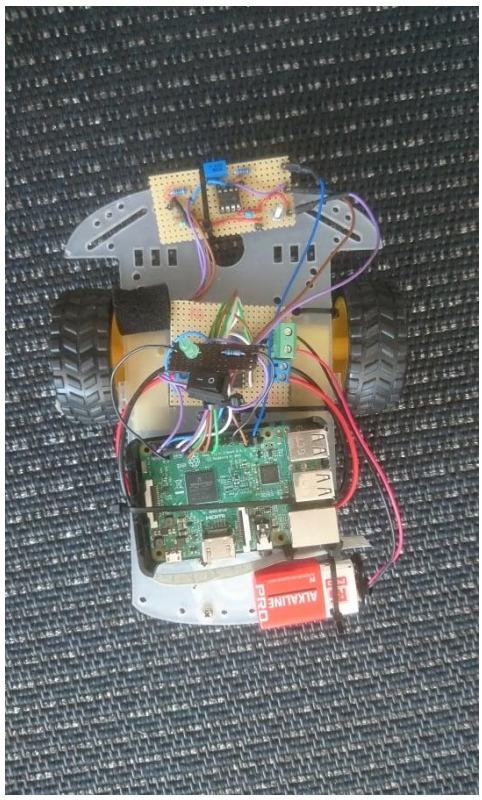


Figure 21: Top-down view of one of the robots

2. Tests

The tests shown were chosen according to the objectives of the project:

Test number	Test Holder	Reason	Expected outcome	Actual outcome	Pass/ Fail
1	Dainius	Test blinking a LED connected to the Raspberry using "Blink diode" command	LED blinks a couple of times	LED blinks a couple of times	Pass
2	Dainius	Check if all voice control keywords perform their intended function	Each voice command works properly (for example, keyword "Drive" drives both motors forward)	All voice commands work properly, but they tend to execute at random or are sometimes hard to activate	Partial pass
3	Juan	See if the software is up and running by blinking the LED after booting the Raspberry up and therefore ready to execute voice commands	If the software is running, then the LED will blink before giving voice command	The LED blinked which means the device is ready to execute voice commands	pass
4	Dainius & Juan	Test the tachometer RPM measuring in the robot and send data through MQTT	Every 60 seconds, the RPM value is displayed on both the robot's and the hub's terminal	Every 60 seconds, the RPM value is displayed on the robot's terminal. After a small delay, it was also displayed on the hub's terminal	Pass
5	Dainius & Juan	Remotely control a robot car using voice control from the hub device	After a small delay, the voice command will be executed	After a small delay, the voice command was executed	Pass

3. Discussion and recommendations

This project was started on January 2018, after finishing work on another project from the last semester. The last project done by this group was ultimately a failure, however, many lessons were learned from it and were utilized in the making of this project:

- Proper time management – the introduction of a periodically updated project plan led to much easier scheduling and time balancing between work and other activities.
- Proper resource management – the project plan let the group make decisions regarding which parts of the project need to be prioritized, and which parts can be returned to at a later

dain0084@edu.eal.dk
juan0314@edu.eal.dk

date. This particular aspect made the development of this project a lot less stressful and a lot more focused.

- More experience since the last project – more knowledge about electronics and project development resulted in the group being able to better prepare for the work to be done.
- Tighter cohesion between group members – each member of the team knew exactly what their specialty and purpose were in the development of the project, and any disagreements or confusion were solved through constant communication.
- More emphasis on practical progress – in the development of the last project too much time was spent on research and theory, while not enough time was spent on design and functional prototype development.

Of course, no project is free of various issues during development:

- Long component arrival times – the group had easy access to most hardware components needed for the project, however, there were some parts that needed to be ordered online and through the lecturers. For example, the robot cars were originally meant to have gyroscopes and other measurement tools for determining rotation and other variables as well as speed. However, the group never received the gyroscopes despite months of waiting, and therefore was forced to seek alternative components or methods. Many components had to be borrowed, and some (like the tachometer) had to be quickly built from scratch using available parts.
- Small development team – only two people were in the group instead of the usual 3-4 members from other groups, therefore the scale and budget of the project had to be adjusted when brainstorming and developing it.
- Scope creep – the periodic addition of new features for a project can significantly prolong its development and even cause it to fail its intended goals. It is most likely evident from the Progress chapter that scope creep was also present in this project's development, as it was originally designed to have speech recognition software in the PC as opposed to the Raspberry Pi, for instance. Fortunately, the group has managed to implement every milestone they set out for the project, but it did result in slightly more difficult project management and timeline keeping.

Despite these issues, this project is considered to be a success, as all objectives initially set out by the group were met, producing fairly impressive results. In addition, the project has further development potential, allowing the group to continue working on it later if they wish.

If this project were to be continued, there is a lot of work that could be done in the future:

- A configuration file or script that could set up user-defined keywords and detection thresholds for them, vastly increasing modularity
- Additional measurement tools that could send more data to the "hub" device, including gyroscopes, battery level measurers and GPS
- Code optimization – rewriting the code from scratch to increase performance and simplicity

dain0084@edu.eal.dk
juan0314@edu.eal.dk

- Proper configuration of the touch display – the display was introduced fairly late in the development of the project, meaning that the group had little time to set it up. It should be possible to display the received or sent measurement values using the screen
- Further keyword detection threshold calibration and better-quality microphones, which would lead to much less false positives and missed detections for voice commands
- Development of custom printed circuit boards (PCBs) for the hardware modules
- Testing different motors and power supplies for them in the system
- Wireless or Bluetooth audio input

dain0084@edu.eal.dk
juan0314@edu.eal.dk

V. Conclusion

1: A script using the PocketSphinx speech recognition engine was able to recognize spoken keywords and execute different actions based on what keyword was recognized.

2: The motor sets were able to be powered and controlled using a motor driver board connected to the Raspberry GPIO, which ran the speech recognition script.

3: A tachometer was built, consisting of a reflection sensor, comparator and some passive components. It was able to send a signal to the Raspberry upon change in voltage (different reflection), which increments the RPM value in the Raspberry. This value was then published using MQTT from the device controlling the motors to the hub device.

4: Two robot models were designed with the same hardware and software (Raspberry Pi connected to the motor set and a tachometer), in addition to a hub Raspberry that can receive measurements and send commands to any of the robots using MQTT.

The results of the project have a great potential and future use in many applications (for example, wheelchair control for those unable to move on their own). The use of easily available hardware and software allows for reusability, further additions and easier bug-fixing.

dain0084@edu.eal.dk
juan0314@edu.eal.dk

VI. References

1. Bibliography

Texas Instruments. (1986, September). L293x Quadruple Half-H Drivers. Retrieved from http://www.ti.com/lit/ds/symlink/l293.pdf at June 3, 2018.

Broadcom Corporation. (2012, February). BCM2835 ARM Peripherals. Retrieved from https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf at May 31, 2018.

Texas Instruments. (1999, October). LMx93-N, LM2903-N Low-Power, Low-Offset Voltage, Dual Comparators. Retrieved from http://www.ti.com/lit/ds/symlink/lm393-n.pdf at June 1, 2018.

Vishay Semiconductors. (Revised 2009, August). Reflective Optical Sensor with Transistor Output. Retrieved from https://www.vishay.com/docs/83760/tcrt5000.pdf at June 3, 2018.

Tarun Agarwal. (2015). Basics of Bypass Capacitor, Its Functions and Applications. Retrieved from https://www.elprocus.com/bypass-capacitor-its-functions-and-applications/ at June 2, 2018.

2. Useful links

Information about the Raspberry Pi GPIO electrical specifications:
http://www.mosaic-industries.com/embedded-systems/microcontroller-projects/raspberry-pi/gpio-pin-electrical-specifications

Homepage of the ALSA project:
http://www.alsa-project.org

Raspbian official website:
https://www.raspbian.org/

Images for Raspbian Stretch and Raspbian Stretch Lite:
https://www.raspberrypi.org/downloads/raspbian/

RPi.GPIO documentation:
https://sourceforge.net/p/raspberry-gpio-python/wiki/Home/
https://sourceforge.net/p/raspberry-gpio-python/wiki/Examples/

GitHub link for Sphinxbase:
https://github.com/cmusphinx/sphinxbase

The official CMUSphinx website and GitHub page for PocketSphinx:
https://cmusphinx.github.io/
https://github.com/cmusphinx/pocketsphinx

GitHub link for pocketsphinx-python:
https://github.com/cmusphinx/pocketsphinx-python

SWIG official website:
http://www.swig.org/

MQTT official website:
http://mqtt.org/

Eclipse Paho official website:
https://www.eclipse.org/paho/

Source code for the Paho MQTT Python client library:
https://pypi.org/project/paho-mqtt/

Putty official website:
https://www.chiark.greenend.org.uk/~sgtatham/putty/