The Automated Bellhop

Develop Intelligent Dynamic Mechatronic Systems
Semester Project Report

by

Catherine Beryl Basson
Wojciech Piotr Zapotoczny
Mathias Skouman Völcker
Vladislav Borodich
Piotr Chromiński

Supervised by
Kasper Paasch

Contents

1 Introduction

The main scope of this project is to properly implement user requirements, as well as to improve the groups understanding of risk management and the overall development of Mechatronic devices. The idea behind this project is to develop autonomous vehicles in order to reduce human error, and improve efficiency. This will include the use of advanced sensors and control strategies in order to provide obstacle/collision avoidance.

Research and brainstorming has been done on activities which require very little or no creative thinking or decision making from the employee and it has been concluded that many jobs could easily be done automatically, with no change or improvement in service, price and speed. Mainly, there was focus on the service industry where many people are employed with transport and delivery, especially small and light objects, as heavier transport is already automated. The problem that this project is trying to solve is that manual labour often is the only option, even where there is no particular advantage in having a human do the job. Examples of such could be jobs that require following only direct instructions, and where decision rarely or never need to be made. Often these jobs also have very uneven workloads that often include a lot of waiting around by the employee, and suddenly switching to busy periods. This kind of work is expensive in that a person has to get paid full time, despite only working for some of the time. The work is

not necessarily pleasant for the employee either, as one may spend much time with nothing to do, even though the employer would expect work to get done. Additionally, a problem may arise in having unevenly-spread periods of heavy workload, as an employee is always expected to be ready to do a task immediately, regardless of if more tasks comes up at the same time. This workflow can easily lead to stress, and it is often not the fastest option due to the substantial amount of work which varies during a day.

This can be seen, for example, in a hotel. Some hotels offer 24hr room service, where it is expected that there would be an employee available to deliver room service at any time during the day. It can be imagined that this has varying workflow and intensity, as more calls to room service may be made during the day particularly around meal times than in the nights / early morning hours. Especially in the early and late hours, these bellhop jobs can be replaced with automated robots, which would deliver the room service request to the hotel customer on demand.

The approach that Group 6 is taking to this project will be to develop an autonomous vehicle that will be able to deliver small items/packages. The main focus of this project is to have a vehicle autonomously move across a floor a building, while avoiding obstacles and following user defined path. The advantage of the robot is that it can provide cheaper, safer and faster service in indoor delivery industry.

2 Problem Formulation

The project will be designed as a multi-purpose robot that easily can be reconfigured to solve other similar tasks, however the focus setting for this design will be hotel room service. Here the problem we solve is that, originally, someone would need to be ready to deliver room service 24 hours a day, even though there may be no one using room service in the early or late hours. This work results in employees sitting around, which is unsatisfying work for the employee, and too expensive to the employer. If the room service delivery were automated, either a receptionist, or someone from the kitchen (who would need to be working at any hour any way in order to make the food requested in room service), can handle the packaging without leaving their own workstation, and therefore eliminating the use of an extra employee.

For hotels currently not offering 24hr room service, perhaps because it is too expensive to hire someone if the room service demand is too low, this product will be an opportunity to improve the hotels service at a relatively low cost.

The main problems that will be faced through replacing manual labour in this situation is indoor navigation, safety in regards to moving around people, as well as keeping the goods safe. There needs to be a way to move around inside through potentially narrow hallways, where a GPS does not work. Ideally, there would be successful navigation without altering the environment more than needed, which means that the existing environment would be used to navigate, rather than, for example, painting lines on the floor for the robot to follow.

The safety issues that come with driving an automated robot around people will also need to be addressed. This includes keeping the people safe, either by not driving into

anyone, or by not driving too fast. It is also important to keep the robot safe from people pushing it around, or trying to take the goods that it is transporting.

Team Integration and Overall Requirements

- Select leaders for various responsibilities.
- Agree on basic rules of conduct.
- Form milestones and goals.

Research and Concept Design

- Consider different sensor modules possible for the vehicle.
- Plan the movement of the vehicle and consider the terrain will it focus on movement over a smooth terrain (such as tiled or laminated floors) or over a rough terrain (such as carpets)? What kind of mechanical components will be required to make it move over this terrain?
- Consider environmental implications that could limit certain aspects or uses of the device (e.g. Can it be used near pools and on wet floors?)
- Look into ways of storing data on movement, mapping, and requests.
- Look into the availability of different microcontrollers or single-board computers that may suit the needs of this project better, keeping in mind and researching some methods that may be needed in order to communicate between them.

Mechanical Development

- Decide on the size of the vehicle.
- Consider the type of drive that needs to be used.
- Determine a method for keeping track of the speed of the vehicle (i.e. encoder).
- Decide on the materials to be used, considering both cost and efficiency/lifetime.
- Create 3D models of the vehicle using a CAD application (i.e. *Autodesk Inventor* or *NX*), as well as sketches, technical drawings with tolerances, and exploded and assembly drawings.
- Consider how to balance the vehicle, and keeping the contents in place, both when stationary and when moving.

Embedded/Electronic Development

- Consider methods for interpreting data from a defined floor plan (mapping).
- Test the sensors and consider applying a filter in order to regulate the data input.
- Make a final decision on the appropriate sensors to be used for this vehicle. For this prototype:
  - 9× ultrasonic sensors to be placed around the perimeter of the vehicle.
  - 2× optical switch to be used alongside an encoder to measure the RPM of the wheels, and thereby the speed of the vehicle.
- Choose microcontrollers/single-board computers and determine the tasks that each one will be assigned. For this prototype:
  - *Arduino Nano* microcontroller for user input, and displaying information (buttons, LCD, etc.).
  - *Arduino Nano* microcontroller for collecting data from the ultrasonic sensors.
  - *Arduino Nano* microcontroller for controlling the motors, and for collecting data from the optical switch.
  - *Raspberry Pi ver. 3 Model B* acting as the brain of the device, which handles requests from the users, handles the data given by the Arduinos, and communicates between the microcontrollers.
- Consider how the vehicle will obtain requests from a user, and how it will use this information. For this prototype:
  - TCP communication will be used to send information of both order and room number from a hotel customer to the kitchen, who will then relay the room number information to the vehicle.

Testing and Assembly

- Perform regular tests throughout the different phases, in order to identify and tackle any problems that may arise.
- Test the movement of the motors.
- Test whether or not the sensors and components interfere with each other.
- Test if any of the electronic components used, such as a multiplexer, add some delay to the collection or relay of information, in order to know whether or not this needs to be compensated for in the software design of this device.
- Fix any problems where the tests provided negative results. This can be done by adding additional components/isolation (for example with sensors that suffer from interference), or by coming up with a different solution entirely.

Documentation

- Document the process and progress include photos of both the steps towards and the completion of milestones and goals.
- Reference any problems that have been solved or unsolved throughout the process, regardless of their inclusion status in the prototype.
- Include the testing methodology, processes, and results.
- Integrate reflections into the documentation Why was one component chosen over another? What worked well and why? What are some considerations for future renditions of this project?
- Include a bill of materials in order to keep track of the budget.

Delimitations and Requirements

- Additional materials may not exceed 1000 DKK.
- The vehicle will not be required or suited to drive outdoors, or on any bumpy terrain.
- The speed of the vehicle must remain low.
- The prototype of the vehicle will not travel to more than one room/location per journey.
- Due to the narrowness of the hallways of hotels, the vehicle will not avoid obstacles by driving around them. In order to prevent collision, the prototype will stop when something stands in its way.

Need-To-Haves

- Collision avoidance.
- SOS system for if the vehicle gets stuck or has stopped due to a possible collision for a long period of time.
- Rechargeable batteries.
- Movement precision.
- Basic user interface (i.e. LCD screen, and button for confirmation).
- PCB of minimum 3cm×3cm.

Nice-To-Haves and Future Additions

- Live feedback.
- Floor map that the device can use to determine the path to the location in an automated fashion.
- Camera for some sort of image detection (for avoiding obstacles and/or for interpreting location by use of strategically placed QR codes).
- Hands-free shelf opening/closing (for goods that are transported).

3 Mechanical Design

TODO WOJCIECH

4 Electornics

The electronic system in the Automated Bellhop is split into six main groups (shown in Fig. 1):

Control	This group contains the processing power of the Bellhop. A Raspberry Pi and three Arduinos connected via USB cables.
User Interface	This is the interface between the user and the Bellhop. Containing A LCD Display and a driver for it, as well as a button for User input.
Ultrasonic	Containing up to 16 ultrasonic sensors, and two high speed multiplexers/demultiplexes to drive these sensors.
Drawer	The two motors used to move the drawer in and out, as well as the 3 hall-effect sensors used to determine the drawer position.
Drive	The two motors connected to the wheels and the rotary encoders on these wheels.
Power Supply	The power management system containing battery, AUX power connector battery charger connecter and switch to switch between the battery and AUX.

Control

The control system consists of one Raspberry Pi and three Arduinos. In the current state of the Bellhop this is not needed, since only the Sensors and Drive Arduinos are used. The reason for the current configuration is that not whole functionality of the robot has been implemented due to time restrictions. The processing power of the Raspberry Pi is needed for the mapping of the hallways the Bellhop drives around.

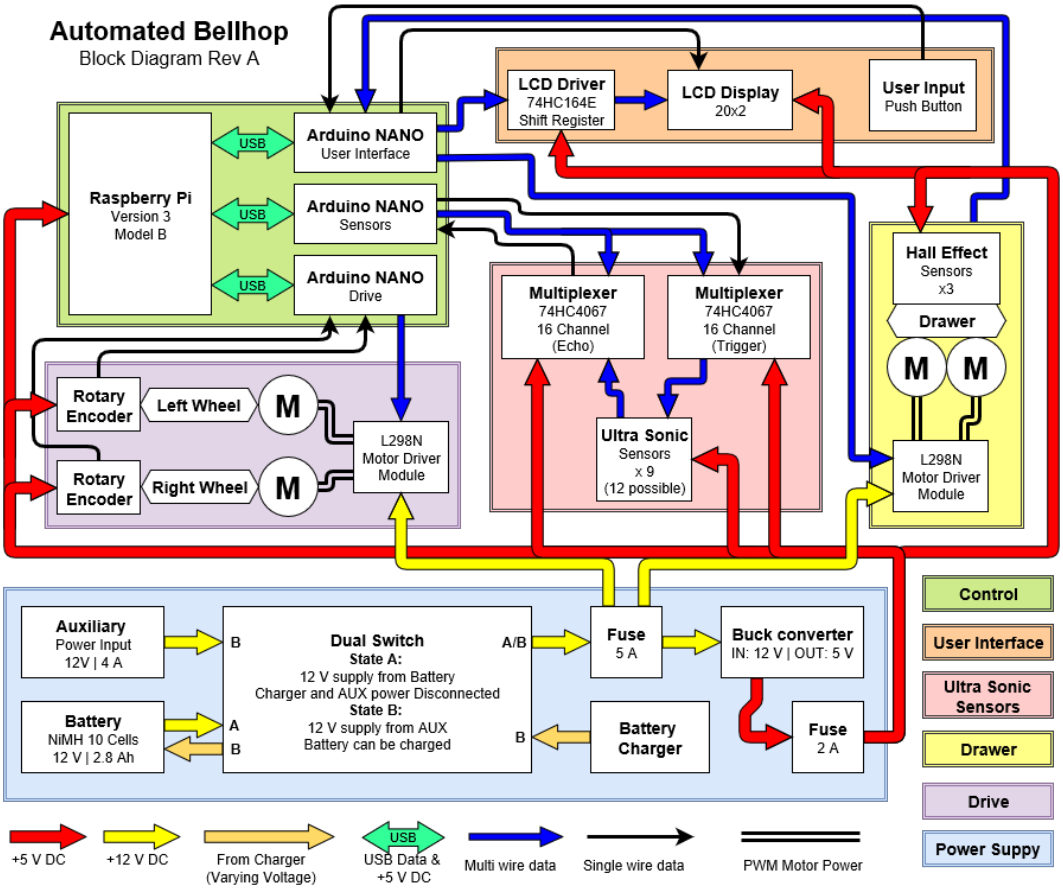

Figure 1: Block diagram of electronics

Each of the three Arduinos has a specific task. The decision to use three seperate units was made because they are cheap and allow for fast prototyping. The Sensor Arduino samples data from the ultrasonic sensors upon request from the Raspberry Pi. This Arduino is reserved for this purpose because ultrasonic ranging requires a pause (waiting for the echo) in execution of the code for accurate measurement, which would disrupt other logic that should otherwhise be executed. The Drive Arduino controls the two driving wheels and the corresponding RPM measurements from the encoders. This must always be ready to alter direction or stop if something gets in its way. As a safety measure, all non-drive essential operation has been removed from this Arduino—to be completely sure it is always ready to react. This leaves the user interaction. Communication with the user, as well as controlling the drawer that should only open when delivering to the right person is handled by one last–User Interface Arduino.

User Interface

The User interaction on the Bellhop is the simplest of the three. It only contains 2 sets of a LCD display and a single button. The 2 sets are located on each side of the Bellhop and allow operation from both sides. The LCDs display the same information, and are connected to the same driver. The LCD backlights are controlled separately to indicate what side of the Bellhop is active. The button is also read separately so the button on the inactive side can be ignored. The LCDs are driven with a *74HC164E* serial-in parallel-out shift register to get the 8 pin parallel data bus needed to drive the LCDs.

The User Interface can be so simplistic because its only purpose is simple communication with the recipient about what is delivered, and if it has been removed. All other communicating, such as destinations etc. is done over a Wi-Fi connection.

Ultrasonic Sensors

To drive the nine ultrasonic sensors (shown in Fig. 2) used in the Bellhop two high speed multiplexers/demultiplexes (*74HC4067*) are used. One for multiplexing the trigger signal to the correct sensor, and the other for demultiplexing the echo to response pin. The sensor is selected using a 4-pin input on *74HC4067* (`S0`–`S3` pins [NXP(2016)]) to set a binary address. The same address pins from the Arduino are connected to both multiplexers, as the relevant response will always be coming from the same sensor that was triggered.

Figure 2: *HC-SR04* ultrasonic sensor used.

Using the multiplexers all nine sensors (up to 16 possible with the multiplexer) are driven using only six pins instead of the 18 that would have been needed to drive each sensor directly. A method that was considered to reduce the pin count was to use a common echo or trigger pin, either trigger only one sensor and listen for response on all of them, or trigger all of them and only listen for response from one at a time. This would be viable in some applications, but could result in crosstalk, where a signal bounces and gets read by a sensor that has not sent out the signal. It would also only bring the pin count down to 10.

Drawer

The drawer uses two DC motors to allow it to open fully on both sides. The DC motors are controlled via a *L294N* bases dual motor control module. This allows the motors to have direction and speed controlled using PWM signals. To determine when the drawer is in the right position, three hall-effect sensors are positioned in the frame, and 3 magnets on the drawer. Two mark the outer positions of both sides, and one marks the centre in the closed position.

Drive

The Drive consists of two geared DC motors, connected to a wheel each. On the axel between each motor and wheel there is an encoder disc that together with an optical switch, used to determine RMP of each wheel independently.

Encoders

For measuring RPM of our wheels, a slotted optical switch and a disc with slits along the circumference is used as a rotary encoder for each drive wheel. The optical switch needs an op-amp to convert the Analog output into a square wave signal that the Arduino can read.

Simply measuring time between rising edges on the square wave the Arduino can calculate the RPM only by knowing the number of slits on the disc and the speed by knowing the circumference of the wheel.

To hold the optical switch in place it is mounted to a small PCB together with the op-amp and resistors needed to make the square wave signal. Having these components as close to the optical switch as possible is important to reduce the noise from the DC motors and the power cables to the DC motors, as shown on the Fig. 3.

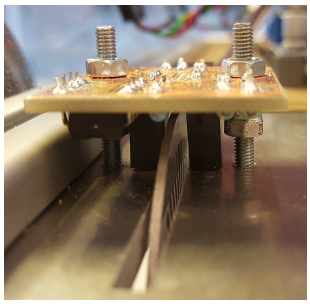

Figure 3: Optical switch PCB mounted on the encoder.

Power Supply

The power distribution system is dependent on a 12V DC input voltage capable of supplying 4A when in full use (Driving on a high friction surface while carrying a load), and 1–2A for testing, depending on the nature of the test. For the main power supply a 10 cell Ni-MH battery pack rated at 12V is used. For testing and continous operation of the microcontrollers doing charging, a 12V AUX power supply can be connected.

Switching between these two power inputs can be done without power shortage, allowing the Raspberry Pi and Arduinos to keep operating without reset when switching. This is achieved by adding an electrolytic capacitor between the 12V rail and GND to remove the few microseconds of power loss coming from switching between the power sources, and smooth out voltage change that can occur as the battery voltage fluctuates doing a discharge cycle and the AUX power supply will give the same voltage regardless of battery voltage. When the power input is set for AUX power, the battery is disconnected from the circuity, and connected to a charging port, allowing it to be charged without removal, or the machine being shut down.

The 12V supply voltage is converted down to the 5V needed for driving the Raspberry Pi, Arduinos, and other logic using a DC-DC switch mode power supply (Buck converter). The Buck converter is unbranded and a datasheet for the module cannot be found. Therefore, the specs are from the datasheet of the main IC. The converter can take an input

voltage of 4–40V and output 1.3–37V [Tex(1999)]. Meaning that the 12V to 5V conversion is well within specification.

Improvements

Turn off 5V for Arduinos doing programming and debugging, to avoid feeding voltage into computer's USB port. Add fan control with temperature measurement to lower noise level when cooling is not necessary. Route communication between Arduino and Raspberry Pi on PCB to reduce cable complexity.

To achieve most of the improvements mentioned above a professionally produced 2 or 4 layer PCB should used. The prototyping PCBs manufactured at SDU provide a great opportunity for testing out hardware, shrinking otherwise very large and complex circuits down by a lot, and mounting SMD chips and other components that have other than 0.1 inch pitch. However, they also have some constraints. The minimum track spacing and track thickness as well as pad and drill sizes are a lot higher on the in-house prototypes compared to etched PCBs. The holes and vias are not plated, meaning that the thru-hole components must be soldered on the correct or sometimes both sides, and that vias must be hand soldered. Lastly, there is no solder mask, silkscreen, or surface finishing to assist with proper component placement and soldering, or to help the final product last longer. Overall, the prototypes are great for simple use cases, but for more complex PSBs with many and/or very small components, they are impractical to use.

	Prototype PCB at SDU	Professionally etched PCB
Layers	1–2	1–4 or more
Thru-hole	No	Yes
Min track width	12 mil	6 mil (4 mil[1])
Min track spacing	12 mil	6 mil (4 mil)
Min drill size	0.6mm	0.3mm (0.2 mm)
Surface finishing	None (exposed copper)	*HASL*[2] lead or lead-free (gold)
Specific to this PCB		
Size	140×100mm	140×100mm[3]
Layers	1	2
Price excl. shipping	140DKK (1 unit)	60USD ≈425DKK (5 units)
Price incl. shipping	140DKK (1 unit)	35USD ≈250DKK (5 units)

Table 1: PCB comparison.

[1] The best possible but also more expensive options, these options are not used in the price comparison.

[2] *HASL* surface finishing refers to Hot Air Surface Levelling which uses a solder bath, and hot air jets to plate all copper connections with solder, this makes soldering easier, and protects the copper from corroding.

[3] As a double-sided PCB optimized for the edging minimum specifications have not been routed, it is impossible to tell the exact size of it. However, as all tracks and vias will be smaller and the routing can be done tighter do to the option of jumping layer a lot easier, it will definitevly not be higher. In the price comparison PCBs of the same size are used.

Values and prices from above table comes from `http://www.pcbway.com/` for the Professionally edged PCB and Mathias Skouman Vlcker how works with the PCB CNC machine at SDU for the limits on that machine. Price for PCB made at SDU:

$$\text{Price} = \text{PCB area} \cdot \frac{\text{DKK}}{cm^2} \cdot \text{Layers}$$

which for the main PCB is

$$\text{Price} = 140cm^2 \cdot \frac{\text{DKK}}{cm^2} \cdot 1 = 140\text{DKK}$$

5 USART Communication

Effective communication between the Raspberry Pi and Arduino slaves means that the reception of commands should not interfere with other logic on the slave. A simple implementation of USART communication on a Arduino uses `scanf()` for input of data, as shown in Fig. 4. This method could not be used because the `scanf()` function waits until

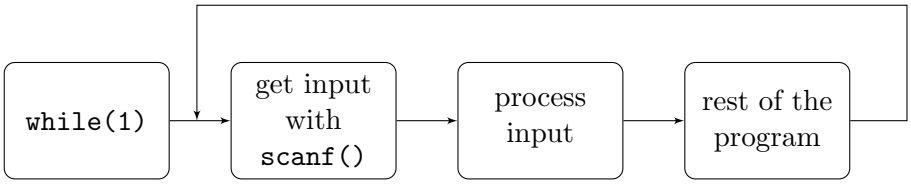

Figure 4: Flowchart of trivial USART data reception.

data comes through the `stdin` stream, which stops the execution of the program. Therefore, the communication needed to be driven by interrupts. The *ATmega328p*'s `RXCIE0` flag is set when there are unread data in the USART recieve buffer, which is cleared when the `UDR0` register is accessed [Atm(2015)], this means that incoming data needs to be processed one byte at a time. The commands recieved needed to be longer than one byte because it allows for easier command decomposition. Example program snippet showing use of asynchronous data reception inside the main loop is shown below:

```
1  if (usart.flag) {
       /* usart_buff is volatile so input needs to be buffered again */
       cli();
       strcpy((char *) input_buff, (char *) usart.buff);
       sei();
6
       switch (input_buff[0]) {
           /* id */
       case '1':
           printf("0\n");
11         break;
           /* more commands */
       }
   }
```

The interrupts are disabled with `cli()`, and `strcpy()` is used because the data in `input_buffer` can change virtually at any time, which would results in corruption of a command. To ensure seamless operation a simple communication protocol was used, each command consists of one line terminated with `\n`. Examples:

`1`	Identify, answer depends on the reciving Arduino
`p1`	Ping: Send data from 1st ultrasonic sensor
`m1Hello`	UI: Write `Hello` on the 1st line of the LCD
`m2World`	UI: Write `World` on the 2nd line of the LCD
`L0`	Drive: Stop the left motor
`Rp5`	Drive: Set the right motor rpm to 5

Error Handling

TODO Cat

6 Network Communication

There are two protocols being used to communicate wirelessly with the Raspberry Pi; one of which is SSH and mainly used for development, and the other is TCP, used between the customer and the Raspberry Pi. Both of these protocols communicate over the network, and are used to enable a nearly hands-free control of the robot from both a development and user perspective. Due to the nature of the robot, it was desired that it be able to receive commands wirelessly, and also, in the hotel setting, it is imagined that allowing the customer to send requests to the Bellhop from the comfort of their room would be more practical. SSH was therefore used to login to the Raspberry Pi, whereas TCP was used as the protocol within the user interface that the hotel customer would interact with.

TCP

TCP communication was a commonly suggested protocol that allowed for the desired outcome/user interface between a hotel customer and the kitchen/Automated Bellhop. Ideally, the hotel customer would (from a computer, phone, or TV) be able to send a request for room service with their order and their room number, which would then be processed and handled with collaboration between the kitchen and the Automated Bellhop. As it is imagined that high-end hotels would likely be the first to make use of this device, based on personal experiefnce it has been noticed that these hotels typically would have Smart TVs or similar that would be able to run an application in order to fulfil this[4].

[4]This can be altered based on hotel requirements, and may be modified to allow the hotel customer to call in their order, where an employee would then relay the required information to the kitchen and to the Bellhop through any communication protocol. This, however, would then potentially require an additional employee.

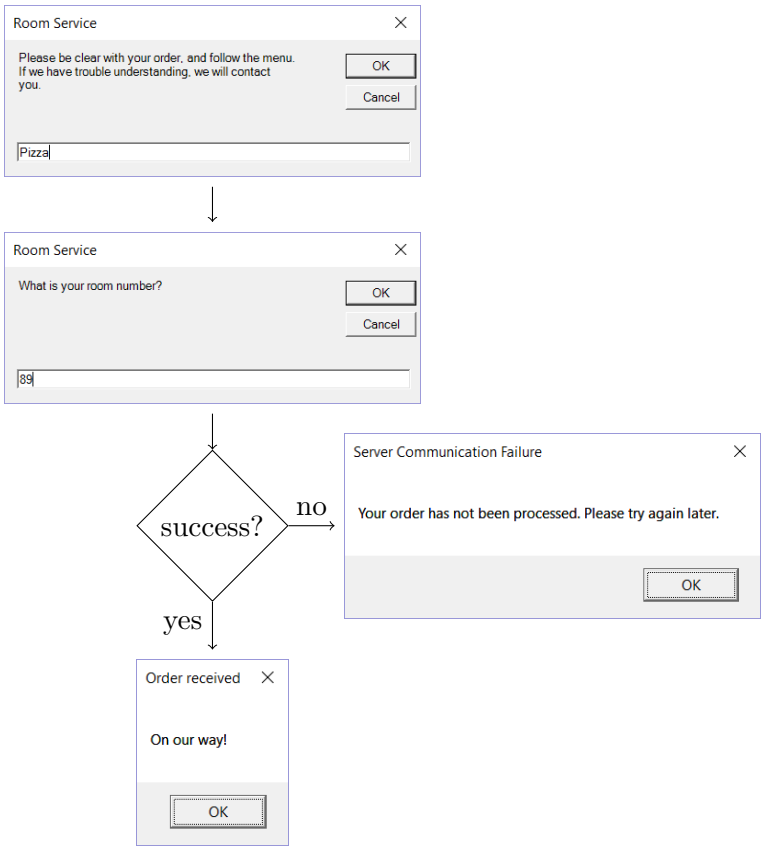

Figure 5: Flowchart of the user's application.

The TCP communication is set up in the following way[5]:

$$\text{Client} \overset{\text{App}}{\Longrightarrow} \text{Kitchen Server} \overset{\text{TCP}}{\Longrightarrow} \text{Raspberry Pi}$$

Where the kitchen would act as the main server that relays data and confirms that communication is successful. In the setup created for the current prototype, the hotel customer would enter their order and room number separately, as shown in Fig. 5. Both the order and the room number are sent to the kitchen, where those working there would be able to see this information (Fig. 6), and the kitchen sends the room number to the Raspberry Pi setup (Fig. 6).

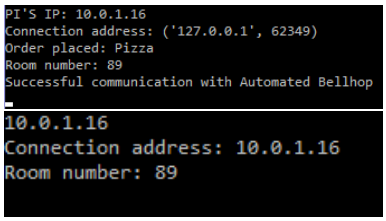

Figure 6: Informations seen by the kitchen and the Raspberry Pi.

In the current state of the prototype, this application was not able to be fully implemented. With a fully functional drive and map, the Raspberry Pi would ideally be

[5] As this example was demonstrated locally, the IP address stays the same between them

able to store the received room number, know its location on the map, and would be able to drive to this location on command once the kitchen employee has confirmed that the order is stored on the Bellhop.

Problems

The following describes some problems that were encountered in relation to network communication during the development of this device, as well as some possible solutions that were used, and otherwise could be used in different situations.

Both TCP and SSH communication require knowing the IP address of the device that is to be communicated with. In a setting where devices are assigned static IP addresses, this is not a big issue, as the IP address would be set once in the code or known by memory for a user, however when the device is assigned a different IP upon each connection, this may be problematic. One simple workaround is to set up a hostname for the Bellhop, which would allow for connection to be established with an unchanging name (i.e. bellhop.local). Provided that the network over which the communication is being established allows for resolvable hostnames, this would be an ideal recommendation for a user of this product. Alternatively, the device could be set up to request a static IP at every connection, however this would also need to be allowed in the network settings.

Neither requesting a static IP address nor configuring a resolvable hostname, however, were not successful given the network settings at SDU. Two options were presented: to set up a personal wireless network, or to use a VPN service, such as *Hamachi* [ham(2017)]. Setting up a personal wireless network would have been the ideal solution given the situation, as it would then be possible to set a static IP or a resolvable hostname, however using the *Hamachi* VPN service was chosen.

In a situation where *Hamachi* is used, it is imagined that both the computer from which the Raspberry Pi is being accessed and the Raspberry Pi itself would be on the same network. Then, a static IP is given within the *Hamachi* network, as well as successful functionality of hostnames, that can be used for SSH and TCP communication. However, there is a limit to the number of devices that can be registered on a network before a subscription service needs to be paid for. This was not a problem during the development of the prototype, however could potentially end up unnecessarily costly if it were to be used in a hotel. Additionally, using a VPN service such as *Hamachi* poses some security risks, as it is then easier for the device to be accessed from outside of the Wi-Fi network. In a hotel setting, this could be a problem, as it would provide the possibility for someone with no relation to the hotel to gain access to the robot device, and thereby potentially some sensitive information that may be stored there. While this is already a potential risk by having the robot run on a network within a hotel, where the guests will in theory be able to access the robot, there is no need to add to this risk by allowing access from essentially all around the world to the robot. This should be considered if a customer should choose to set up a VPN service in order to work with the robot.

During the development of the robot, the main problem was the loading time for the Raspberry Pi to be logged in to the *Hamachi* network. While it was set up to log in to *Hamachi* upon boot, it still took up to five minutes before the connection was successful,

despite nearly immediately connecting to the Wi-Fi network. In order to keep track of this, a simple LED was set up on the Raspberry Pi GPIO pins that allowed for a visual indication that could display the status of the network connection, using a python script which would also begin at startup, and also continue to run throughout the use of the Raspberry Pi. The LED remained off until the Raspberry Pi had connected to the Wi-Fi, where it would instead start blinking. When the Raspberry Pi has successfully logged in to *Hamachi*, the LED remains on. In the final version of this robot, this would also be implemented in order to indicate when the Automated Bellhop would be ready to serve. However, nearing the final weeks of development, the time delay was detrimental to the development process, as whenever it was required to restart the Raspberry Pi, there would be an unwanted amount of time before development could continue running. Therefore, it was instead set up that upon connection to a Wi-Fi network, the Raspberry Pi (through the use of a connected Arduino Nano microcontroller) would print its IP address on an LCD screen. This was a successful solution during the development of the robot, however should not be used in the final product.

Many of the problems that were faced in relation to the communication with the device (both through SSH and TCP) were due to some limitations set by the environment in which the robot was being developed. Though there were workarounds that were used for the purpose of prototyping, the problems shed light on some possible requirements or recommendations that would allow for the device to be setup and run smoothly in a hotel setting – mainly for the hotel to ensure the network can allow for a device to request a static IP, or for the network to allow for resolvable hostnames.

7 Arduino Nano Microcontrollers

The final of this product would ideally make use of three *Arduino Nano* microcontrollers: one to control the user interface on the device (i.e. LCD and buttons), one to control and collect data from the distance sensors, and one to control the drive of the robot. The three microcontrollers are connected via USB to the Raspberry Pi, in order to send and receive any necessary data in order to function in the desired. Additionally, by having the microcontrollers connected via USB, it is then easy for if the program on any one of them needed to be modified, as this could be done through SSH communication to the Raspberry Pi[6].

User Interface Arduino

The User Interface includes all interactions with the Bellhop, such as: displaying messages on the two LCD screens, input from the buttons, and controlling the drawer. Currently, it is only used to display Raspberry Pi's ip address on the LCDs.

[6]In the current state of the prototype, four Arduino Nano microcontrollers are being used, as one is being used simply to print the IP address of the Raspberry Pi to an LCD.

Sensors Arduino

The robot has been set up with eight ultrasonic sensors (*HC-SR04*) around its perimeter; three in the front, two on either side, and two in the back. Those on the sides would be used in order to determine how close the robot is driving to a wall, for example, which the Raspberry Pi would then interpret and adjust the turning of the robot accordingly. The ultrasonic sensors set up at the front of the device are in order to detect an oncoming collision, which would then cause the robot to stop its movement until the obstruction is out of the way. The ultrasonic sensors at the back are setup for the use with the map, in order to help with determining the space around the device, and distance to any walls or obstructions on turns. Sensors initially were setup in a way where all eight shared a trigger pin on the microcontroller, and then each had a dedicated echo pin. However, it was decided that this may cause some interferance or inconsistencies in measurements, as it may happen that an ultrasonic sensor picks up the echo from a different sensor than itself. Therefore, the setup was changed to make use of two multiplexers; one to handle the trigger pins, and another to handle the echo pins. This setup is done by writing the ID of the ultrasonic sensor in use into the `PORTB` register, which is done by the `multiplex()` function:

```
1 void multiplex(uint8_t id)
  {
          PORTB &= ~(0xFF);
          PORTB |= id;
  }
```

The distance measured by an ultrasonic sensor is determined in the following way: When the microcontroller receives a request to read from a specific sensor, it will send a pulse through the trigger pin on the desired ultrasonic sensor. When the echo pin goes high (which is done automatically by the ultrasonic sensor after some small delay from the time the pulse is sent through the trigger pin), a timer begins, and the overflows on this timer are counted by adding 255 to the count variable. The echo pin goes low when it has received a signal back, and this stops the timer, and the remaining counts are added. Then, this count value is converted to distance (in cm) using the following equation:

$$\texttt{distance} = \texttt{(double) ping.echo} \cdot \frac{\texttt{US\_PER\_CLK}}{\texttt{US\_PER\_CM\_2}}$$

where `US_PER_CLK` is the amount of microseconds that pass in one clock cycle on the microcontroller (set to 16MHz), `US_PER_CM_2` is how many microseconds pass per centimeter travelled by sound multiplied by 2, and `ping.echo` is the number of counts. These macros are defined at the beginning of the program, in case they need to be changed for precision.

Within the program, it is also determined whether or not the maximum amount of counts have passed before the echo has come back, known as `PING_MAX`, which is set to be equivalent to a reading for 400cm, as the datasheet describes this being the maximum accurate reading [ELE(2016)]. Once the count exceeds `PING_MAX`, the time stops, and the program sends back an error. However, later in the development, it was noticed that there

is some delay between when the timer exceeds `PING_MAX` and when the ultrasonic sensor can be used again—about 200ms. This is due to the nature of the ultrasonic sensor itself, as its internal timeout for having the echo pin go low is 200ms, which is much longer than what it would take for the echo to bounce back from 400cm (about 23ms). This causes a substantial delay if it is desired to read from one ultrasonic sensor multiple times in a short timeframe (where one or some readings, perhaps, does not come back) as the trigger can only send a pulse once the echo pin has toggled down. While this should not be a problem in the final make of the device, it did raise some problems in testing.

There are some ways that this problem could be fixed or worked around, firstly to consider the use of different kinds of sensors that can either have the echo pin be manually set down, or that uses a different method to collect data where this problem does not arise[7]. Alternatively, a work around could be to manually cut power to the ultrasonic sensor after a predetermined amount of time has passed (i.e. that of `PING_MAX`), through the use of mosfet transistors to clear the sensor's echo pin, by turning it off and on.

Drive Arduino

For driving the Bellhop we use two *919D501LN* motors which are controlled by a *LN298N* dual channel H-Bridge. Controlling both engines rotation and direction is done with 4 pins and 2 PWM signals. A frequency of 40kHz was chosen for the PWM, as it proved to be the most effective in testing, also the limit for the H-bridge is 40kHz [ST(2000)]. Having 2 independent drive wheels made it difficult to drive straight, as each engine outputs slightly different power, for instance when both engines are set to 100% power, the Bellhop would steer left, we try to correct that problem with use of feedback from encoders to control the RPM.

Encoders

Encoders are needed because without them there is no way to gauge how much the Bellhop has travelled on each of the wheels. First of all, a sensor is needed that would do the measure rotation of the encoders, available in E-Lab is the *OPB866T55* optical switch. One of the problems of using this switch directly with an Arduino is that the voltage does not rise immediately when the emitter and collector detect each other, which would lead to uncertainty, to correct this problem an op-amp is used to filter out signals below 2V, and get a more stable reading on the Arduino, for schematic of the

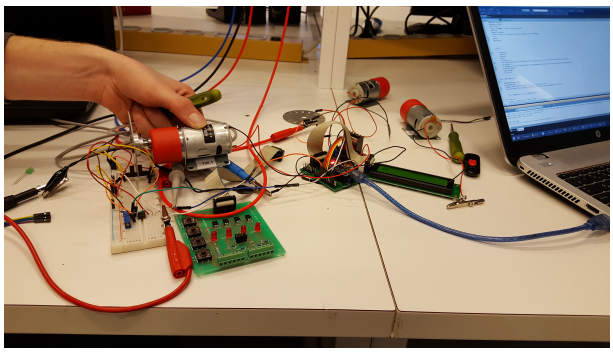

Figure 7: First test of the encoders.

[7]However, it is possible that another problem that is unforeseeable may arise, as no other distance measurement sensors have been tested in this purpose.

switch refer to the Appendix[8]. On Arduino timer inerrupts are used every 0.05ms(20kHz), as getting the number higher resulted in problems with USART communication, as it was interrupting too much, and slowed down the communication of data.

Six encoders were designed with varying slit size, As a test encoder 1 and 6 were water-cut out of stainless steel with, as they were on the extremes of each other. The picture of initial test can be seen on the Fig. 7. The 1st encoder is good at being super fast, having a 3° slit and 3° of spacing, with ≈60 reading per revolution from it, but as tests have shown it is not stable enough as at idle engine RPM(no load) it varied quite a bit ≈4 counts as shown in formula for calculating RPM. This may be attributed to the inaccuracies which are stated below. The 6th encoder has 6° slit and 18° of spacing which proved to be much more consistent, but having only 15 readings per revolution meant that RPM control could not be done very quickly as it had long times in between readings especially at low RPM ≈30. The test for the encoder was that for 5 mins I would observe its time data and see the deviation, of which it had none, it was surprising as the 1st encoder was much more inaccurate. Further testing is required to find the best possible one, which has the best accuracy (readings/rev).

The formula for caluculating the RPM for the first encoder was:

$$\text{RPM} = \frac{1}{x \cdot 0.5 \cdot 10^{-4} \cdot 60} \cdot 60 = \frac{20000}{x}$$

where x is the count of interrups from rising edge to risig edge of the op-amp output, and $0.5 \cdot 10^{-4}$ is the period between interrupts, and 60 is the amount of high to high transitions in one revolution. The 2nd encoder had a formula of RPM $= \frac{80000}{x}$, after simplification.

One thing that can not be completely certain is that what is measured the truly the right RPM, as we there is no tachometer in the SDU workshop that works precisely enough, the one tachometer the workshop has is the one which has flickering light at different frequencies, with which confirmed that the RPM meaured by the Arduino is roughly the same (≈140RPM on the tachometer, and 140RPM on the Arduino). Another precision point would be the water-jet machine, as the holes for the Axle to go through are supposed to be 9mm, but in reality after measuring with caliper they were 8.7mm, which leads to question if the encoders are truly what we designed them to be from edge to edge—24°.

RPM Control

For the RPM control a simple formula which adjusts the duty cycle output was implemented:

$$\texttt{pwmL} = \texttt{pwmL} + (\frac{\texttt{rpmLset}}{2} - \frac{\texttt{rpmLread}}{2})$$

It responds linearly by giving, this formula worked very well with encoder 1 as it's fast readings allowed for quick adjustments to PWM to get the desired RPM, if it was possible power wise. However, it tended to constantly overshoot/undershoot, and setting the RPM at 60 would mean that the wheel RPM would fluctuate between 58–62RPM which was

[8]TODO path to optical-switch-schematic

not accurate enough. With the Encoder 6, it did not work very well, as when the speed suddenly decreased it would overshoot duty cycle by a large margin, and would take a while to stabilize to get the same results as Encoder 1. Another formula which is much simpler incremented PWM by 1, This one was super smooth in its operation, but was not fast enough to react to sudden changes, as it would take a while to get from 30rpm($\approx$40% duty cycle) to 60rpm(100% with load).

Unfortunately, the RPM control was not implemented in the current program as it was causing problems, and was replaced with direct PWM control. One problem of the engines is that even though they can pull 1.4Nm they draw a lot of current and the *L298N* H-Bridge cannot supply it, even at normal operation with no load, the voltage drop was 0.2V. All in all, maybe it would be even better to have 2 encoders on each wheel, one would reliably count the Revolutions, and the other would be used to control RPM.

8 Raspberry Pi

The control of the robot as a whole is being done through the use of a Raspberry Pi 3 Model B, which has been set up to communicate through USART with the three Arduino Nano microcontrollers. This allows for the Raspberry Pi to act as a master in this setup by interpreting the data received from one end, and then issue the necessary commands based on this information to the respective microcontroller.

An example of this communication can be seen when data from the ultrasonic sensors is being collected. The Raspberry Pi will send a request to the Arduino Nano microcontroller that controls the ultrasonic sensors to send back data from specific sensors, for example, those on the right side of the robot. The Raspberry Pi would then process this data, and, based on the outcome, send the appropriate command to the Arduino Nano microcontroller that controls the drive, such as turning more left if the robot is detected to be too close to a wall.

Additionally, the Raspberry Pi has been configured to compile the C codes and then program the microcontrollers directly, so it is unnecessary to disconnect them and compile from a PC. SSH communication was used from a remote computer to connect with the Raspberry Pi over a network in order to update any scripts and programs wirelessly from a computer, without needing to disassemble the robot. This is especially useful for troubleshooting, and when testing the assembled robot, as it was possible to, from a laptop, send commands to the specific microcontrollers in order to see if they respond correctly.

Programming the Arduino microcontrollers is done through the use of a USB cable, similarly to how they would be programmed from a laptop. On the Raspberry Pi, which runs on a Linux based operating system (*Raspbian Jessie Lite*), serial USB devices (i.e. the microcontrollers) are considered as character devices. The kernel assigns a file (in this case: /dev/ttyUSB0, /dev/ttyUSB1, etc.) to the specific microcontrollers based on the order in which they are plugged in[9]. Therefore, as it is possible for the individual *Arduino*

[9]In Windows Systems, this would be similar to what would be considered COM Ports, which are instead hardware-based, and rely on the specific USB port to which the serial USB device is connected.

Nano microcontrollers to be assigned different files on startup, or if they are plugged out then in again, it is necessary for the makefile to determine on to which microcontroller it must upload the code. In order to accomplish this, a Python script, which can be seen in the digital appendix[10], is called within the makefile, where the input is customised based on to which microcontroller the makefile should build. For example, in the makefile for the microcontroller that controls the drive:

```
PORT = $(shell ./usb.py drive 2>/dev/null)
```

with `usb.py` returning `/dev/ttyUSBn` where n is the device number of the Drive Arduino.

Map

Ideally, the Bellhop would travel with assistance of a built-in map, determine the shortest distance to the destination, and recognise if there is an obstruction in front of itself by comparing expected ultrasonic sensor output with actual readings; combined with set rpms and feedback from the encoders a Kalman filter could be use for accurate position estimation. Considering these criteria it was chosen to use *Python 3.5* and a Raspberry Pi due to the ease of developement, interest, and learning outcomes. The map is represented as a list of line coordinates (*x1, x2, y1, y2*), and stored in a `.json` format. During the developement of the map logic a simple GUI program was used to simulate simple Bellhop movements and test algorythms. For the GUI different libraries were considered: *cocos2d* [coc(2017)], *pySDL2* [PyS(2017)], *PyCairo* [cai(2017)], and *Tkinter*; out of these *Tkinter* was chosen because it is a standard python module and it is the simplest use out of the four. The source code for this program can be found in the digital appendix[11].

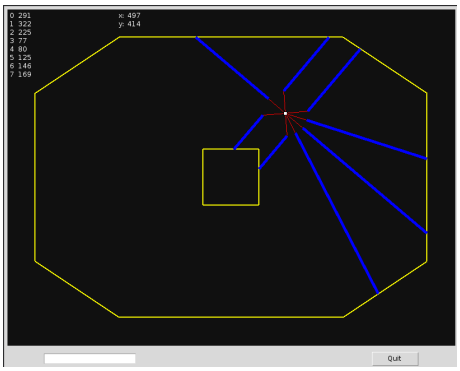

Figure 8: Tkinter map with a Bellhop object, walls shown in yellow, ultrasonic sensor distances in blue, distance from center of the Bellhop to a sensor in red.

The approach for modeling ultrasonic sensors is a simplified version of James L. Crowley's model [Crowley(1989)]. Each sensor is defined by three variables:

`r`	The distance from the center of the Bellhop to the sensor.
`beta`	The orientation of the sensor with respect to the robot's axis.
`gamma`	The Angle from the robot axis to the sensor.

The `Bellhop` class has it's `alpha` angle—orientation with respect to the map, `x` and `y` which are it's estimated position on the map. Currently, there are working algorythms for

[10] `/src/Raspberry/usb.py`
[11] `/src/Raspberry/Map/map.py`

estimating readings from ultrasonic sensors based on sensor input. These are implemented as methods the `Map` class. This logic can be found in the digital appendix[12].

In the developement process unit tests were used, using the Python's unittest [uni(2017)] framework. For each method in `Bellhop` and `Map` classes there is an isolated test which ensures that it's output is correct and all exceptions are caught. This has proven very useful and has found bugs that would not have been caught otherwise[13].

Brain

The main program controling the Bellhop is written in *Python 3.5*, and can be found in the digital appendix[14]. The logic is divided into several classes responsible for interaction with users, motor control, sensor data, etc. It has been done in a way where each class is specialised in one area. For example, the `Ping` class can only control the `Ping` Arduino, or the `Motor` controls one motor. Overview of the classes is shown below:

`Lcd` Controls the `UI` Arduino. Methods for displaying messages on the LCDs, button input etc.

`Ping` Controls the `Ping` Arduino. Reads data from the ultrasonic sensors, could possibly be used also for filtering of data.

`Motor` Controls a motor, sets the rpm or gear.

`Drive` Controls left and right motor, consists of `Motor` instances.

`Brain` Controls the Bellhop, has instances of `Lcd`, `Ping`, `Drive`, and `Map`[15]. Makes use of them to perform more intricate tasks. For instance, keeping constant distance to a wall while checking for collisions.

This structuring allowed for each of the individual functions of the robot to be developed and tested seperately. However, it was not until the Bellhop was assembled when some issues were found, due to the time constrains they could not have been solved. Namely, the $200ms$ delay when an ultrasonic sensor's reading is out of range and urelaiable control of the motor rpm, which made the control of the Bellhop unreliable.

Safety Distance

Safety should be the number one priority for a robot moving where people also walk around. Different types of obstacle avoidance and rerouting of the drive path is preferred over a full stop, but as a last measure if everything else fails, the robot should stop completely before hitting anything or anyone. Therefore, three ultrasonic distance sensors in the front, and 2 in the back are used to detect obstacles in the drive path. If this detect something inside

[12] `/src/Raspberry/Map/bellhop.py`
[13] `/src/Raspberry/Map/tests.py`
[14] `/src/Raspberry/brain.py`
[15] Not implemented in `brain.py`.

the safety distance of the robot, it stop immidiately, and not move until the situation is analysed, and the obstacle has moved or is determined to be stationary, in which case backing away is an option. The safety distance will after all testing is done be dependent on current velocity, but for now it is set to a static value high enough for the robot to stop safely from maximum velocity. Optimally, the safety distance could be defined in two parts: one where the robot must stop instanteniously to avoid collision, and second where it is able to break gently to protect it's shipment from damages.

Currently, a simple collision avoidance is implemented in a form of a method in the `Brain` class:

```
def check_front(self):
    """Check if an object is blocking in front"""
    dist = self.ping.distance(2)
    while dist < self.safety_distance:
        print('something in front {}'.format(dist))
        self.drive.stop()
    self.drive.forward()
```

When the reading on the center front ping sensor is smaller than the acceptable safety distance the Bellhop will stop and wait until the reading is acceptable before continuing. It could easily be extended to include safety distance for each side of the robot. This however does not account for faults to Raspberry, which could result in the motors being stuck in forward gear, and causes collision risk. Therefore, additional system could be implemented where the Drive Arduino would have internal timeout for the motors, which would be reset every time a command is recieved from the Raspberry.

9 Division of Work/Responsibilities

Catherine

- TCP Commuication
- User experience and usability
- Raspberry Pi usability and control (startup, etc.)
- Ping Arduino

Vladislav

- Drive Arduino
- Mechanical design
- Encoders
- Mechanical Workshop Contact person

Piotr

- Map
- Brain
- Ping Arduino
- User Interface Arduino
- USART communication
- E-Lab Contact person

Mathias

- PCB design
- Electronics handling/design

Wojciech

- Mechanical Design

10 Technical Review from a Senior Software Architect

TODO Cat Main:start small!!!! i.e. fewer ultrasonic sensors, stepper motors

11 Conclusion

TODO Cat

12 References

[NXP(2016)] *74HC4067; 74HCT4067 16-channel analog multiplexer/demultiplexer*. NXP Semiconductors, 2016.

[Tex(1999)] *LM2596 SIMPLE SWITCHER Power Converter 150-kHz 3-A Step-Down Voltage Regulator*. Texas Instruments, 11 1999.

[Atm(2015)] *ATmega48A/PA/88A/PA/168A/PA/328/P*. Atmel, 5 2015.

[ham(2017)] Hamachi, 2017. URL `https://www.logmeininc.com/`.

[ELE(2016)] *Ultrasonic Ranging Module HC - SR04*. ELEC Freaks, 2016.

[ST(2000)] *L298*. ST, 2000.

[coc(2017)] cocos2d, 1 2017. URL http://python.cocos2d.org/.

[PyS(2017)] pysdl2, 1 2017. URL https://pysdl2.readthedocs.io/.

[cai(2017)] Pycairo, 1 2017. URL https://cairographics.org/pycairo/.

[Crowley(1989)] James L Crowley. World modeling and position estimation for a mobile robot using ultrasonic ranging. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 674–680. IEEE, 1989.

[uni(2017)] Unit testing framework, 1 2017. URL https://docs.python.org/3.5/library/unittest.html.