|  |
| --- |
|  |
| **Autonomous Maze Mapping and Running Rover** |
| Operation Systems and Embedded Linux, Parallel Programming |
|  |
| **Team 3 / IOSLX4, IPARP4** |
| **5/1/13** |

**Supervisors:**

Ole Rydahl, Henning Haugaard

**Student name Student number Signature**

Johannes Jørgensen 120428

Benjamin Prakash 100100

Ivo Drlje 101108

Daniel Machon 092039

Table of contents

1. Introduction 4

**1.1** **Project start** 5

**1.2** **Problem formulation** 5

**1.3** **Milestone plan** 5

2. Problem analysis 5

**2.1** **Requirements** 6

**2.2** **Analysis** 6

**2.2.1** **Navigation** 6

**2.2.2** **Mapping** 7

**2.2.3** **Path finding** 8

**2.2.4** **Choice of programming language** 8

3. Proposed solution strategy 9

**3.1** **Git** 10

**3.1.1** **Branches** 10

**3.1.2** **Issues** 11

3.2 Resources 11

**3.2.1** **Hardware** 11

**3.2.2** **Software** 12

4. Problem solution 12

**4.1** **Navigation** 14

**4.1.1** **Design** 14

**4.1.2** **Implementation** 14

**4.1.3** **TEST** 14

**4.2** **Mapping** 15

**4.2.1** **Design** 15

4.2.1.1 Position 15

4.2.1.2 Coordinate system 15

4.2.1.3 Mapping algorithm 16

4.2.1.4 Robot / PC - strategy 17

**4.2.2** **Implementation** 18

4.2.2.1 Robot / PC communication 20

**4.2.3** **TEST** 20

**4.3** **Path finding** 20

**4.3.1** **Design** 20

4.3.1.1 Defining the graph 20

4.3.1.2 Robot / pc strategy 23

**4.3.2** **Implementation** 24

4.3.3 TEST 25

5. Conclusion 25

5.1 Process assessment 25

6. Bibliography 26

7. Glossary 26

8. Appendix 27

Appendix 1. Milestone plan 28

Appendix 2. Action item list 28

Appendix 3. Test cases 28

Appendix 4. Hardware diagram 33

Appendix 5. Process diagram 34

Appendix 6. Mapping logic flowchart 35

Appendix 7. Mapping flowchart I dunno 36

Appendix 8. Workflow diagram 37

Appendix 9. Sequence diagram 37

Appendix 10. 38

Appendix 11. 38

Appendix 12. 38

Appendix 13. 38

Appendix 14. 38

Appendix 15. 38

Appendix 16. 38

Appendix 17. 38

Appendix 18. 38

Appendix 19. 38

Appendix 20. 38

Appendix 21. 38

Appendix 22. 38

Appendix 23. 38

**Motor** 38

Stepper Motor 38

8.1.1.1 DC Motor 39

8.1.2 Auto correction input 40

8.1.2.1 Odometry 40

 Mouse 40

 Wheel Encoders 40

8.1.2.2 Sensors 40

8.1.3 Auto correction 42

8.1.3.1 PID 42

8.1.3.2 Iterative 43

8.1.4 Path finding 43

8.1.4.1 Dijkstra 44

8.1.4.2 Flood fill 45

8.1.5 Communication 45

8.1.5.1 Zeroconf 45

8.1.6 Mapping 46

8.1.6.1 Depth first search 46

8.1.6.2 Graph 46

RPi-TMC222 baud rate Documentation 47

Preface

The report consists of two parts, a main part and appendix part. The context of the main part will throughout the report use refers to the appendixes part for illustrations. Included is a CD containing Python files and a word/PDF document of the report.

# Introduction

## **Project start**

This paper documents the development process of the 4th semester project “**Autonomous maze mapping and running rover**”, which is an interdisciplinary project, involving the courses ‘Parallel programming’ and ‘Operating systems and embedded Linux’.

The project comprises the construction of an autonomous rover, using available hardware parts provided by the teachers, along with the development of a software solution to control the rover and map the maze. The software solution also includes the implementation of a client computer that will request the mapped maze, and show it graphically.

Given a multitude of different technologies, software strategies and general development approaches, we as a group will discuss and compare these, to delimitate our project solution and present what we believe is the engineerically correct/best solution to our project requirements- and formulation.

## **Problem formulation**

We want to explore the different approaches to navigate an autonomous robot and find the most appropriate inputs and algorithms to interpret these inputs in order to map a maze and avoid collision with the walls by driving straight. Furthermore we want to generalize the mapped maze into a data-model and a graph for use in path finding as well as a way for the robot to receive this path and drive to the target destination

## **Milestone plan**

The milestone plan is enclosed in the appendices: see Appendix 1

# Problem analysis

## **Requirements**

|  |  |
| --- | --- |
| **Functional Requirements** | |
| **R1** | The Rover must be able to map any maze. |
| **R2** | When the robot has finished mapping it must be able to transfer the map to the PC. |
| **R3** | The PC displays the map in a GUI. |
| **R4** | The PC will be able to find the fastest path between two points in the map. |
| **R5** | When a path has been defined it will be transferred to the robot. |
| **R6** | The robot converts the path into instructions and traverses the path and stop at the target destination. |
| **R7** | During mapping and traversal the rover should record a log of sensor readings and motor orders. |
| **R8** | The robot should be able to autocorrect its traversal to avoid collision. |

|  |  |
| --- | --- |
| **Non Functional Requirements** | |
| **R1** | The robot must use 2 stepper motor. |
| **R2** | The robot must use a Raspberry Pi model B for controlling the unit. |
| **R3** | The robot must use 3 sensors for determining its surroundings. |
| **R4** | The robot must be programmed in the object-oriented language Python. |

## **Analysis**

### **Navigation**

The robot needs to turn using differential drive.

The robot will use sensors in order to detect surroundings.

The robot has to use the sensory inputs to autocorrect its position in the middle of the path.

The robot should be able to log important information about I/O as well as the navigation logic.

The robot needs functionality to calculate its traversed distance.

Identified problems:

|  |  |  |
| --- | --- | --- |
| Subject | Problem | Proposed solution |
| Sensors | The robot needs a way to identify surroundings | IR-rangefinders, ultrasonic |
| Odometry | Distance traversed, current position/direction | Wheel-encoders, mouse-sensors, step-counters |
| Auto correction | The ability to stay center in the corridors | PID, iterative trigonometric corrections |

### **Mapping**

The robot will need to be able to map the maze autonomously.

It has to use an algorithm that takes care of loops and dead-ends but not open spaces when it explores the maze.

The robot needs a data model for storing explored cells in the maze and also a way to transfer this model to the PC.

The PC has to be able to display the maze in a graphical interface as well as take commands from the user.

When the robot has finished mapping the maze it has to stop and wait for further instructions.

The robot has to be able to traverse and map any maze excluding open spaces.

Identified problems:

|  |  |  |
| --- | --- | --- |
| Subject | Problem | Proposed solutions |
| Mapping algorithm | Explore and store the maze without redundancy. | Depth first search, flood fill |
| PC communication | Transfer the maze model to the PC | JSON, Pickles |
| PC GUI | Receive a maze map and display it graphically | Java/Swing, Python/QT |

### **Path finding**

The PC has to be able to find the fastest path between any given two points in the maze.

The PC needs to be able to take user commands and in the end send the path to the robot.

The robot will have the ability to translate the received path into instructions for driving in the maze and stop at the target destination.

The robot has to return to idle state when finished driving a path.

The robot has to be able to transfer its current position to the PC so the PC can use this position as source when finding a path.

The robot has to be able to stop at any given target in the maze including points in the middle of a corridor.

Identified problems:

|  |  |  |
| --- | --- | --- |
| Subject | Problem | Proposed solutions |
| Fastest path algorithm | Find the fastest path between two points in the maze | Dijkstra, a-star, flood fill |

### **Choice of programming language**

The software on the robot has to be object-oriented.

We will to use an interpreted language since development will be faster without cross compiling.

We will focus on a language with a large standard library that runs same code on many platforms.

The language needs to be able to use the I2C bus of the R-Pi.

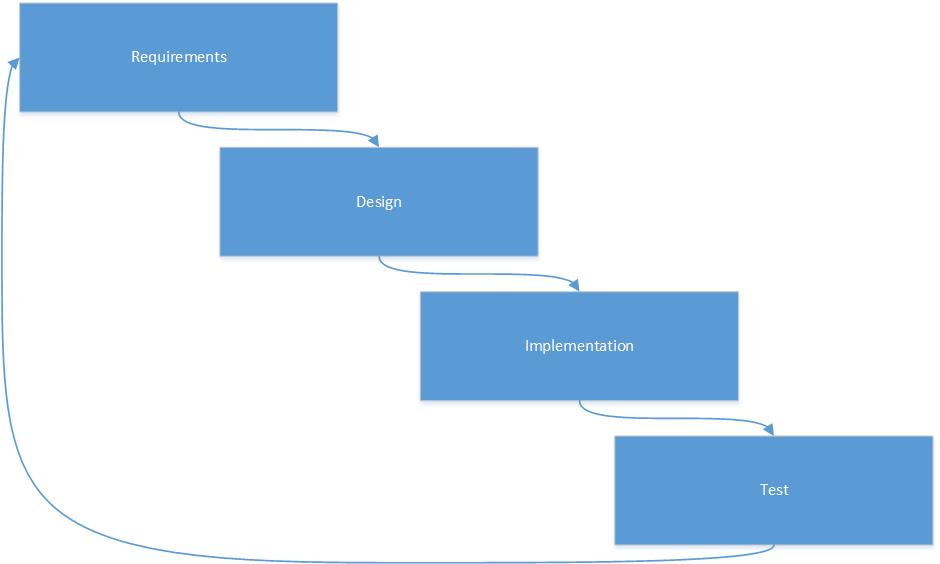
Identified problems:

|  |  |  |
| --- | --- | --- |
| Subject | Problem | Proposed solution |
| Language | Find an OOP language that runs on arm/PC | Python |

# Proposed solution strategy

We plan to use our own version of an iterative deterministic process development system, which mainly focuses on the phases of design, implementation and testing of code.

The development of the robot and code takes foothold in the identified project requirements, where the design of the solution to a requirement is first discussed thoroughly, before taken into the implementation phase.

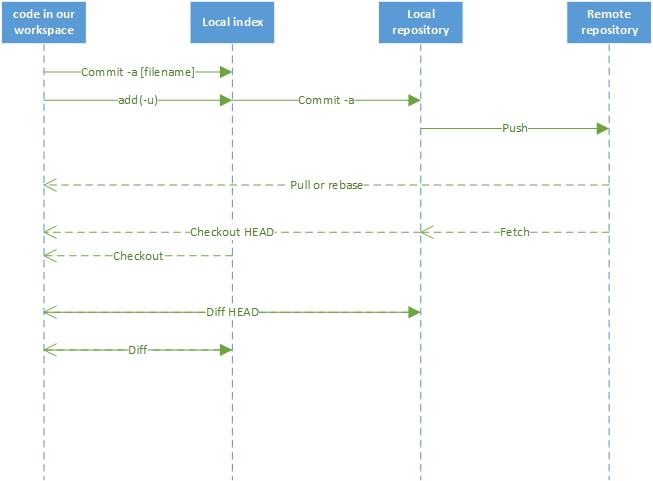


The development process is structured around maintaining a weekly ‘week description’ which summarizes and documents all project activities of the week. This week description will keep track of project- and code development as well as time spent on different problems, in order to backtrack and pinpoint badly selected choices.

## **Git**

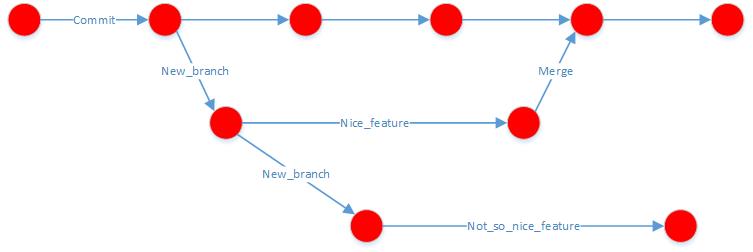
The software solutions to a given problem will be written in Python. All group members will keep their own local workspace synchronized with an up-to-date remote repository, through the use of Git.

Git is a distributed revision control system that allows us to develop project software, without the need to maintain a connection to a common network. Git will let us create a remote repository to contain project code, and provides an excellent version control system, to make sure local and remote code is up-to-date.



### **Branches**

One of the most efficient git functions we plan to use is branching. Branching allows us to create a new branch of code that will extend a current branch (most likely the master branch). This means that if any developer wants to enhance the current code, they can do so by creating a new branch and code their new feature. If this new feature suits the group of developers as a whole – it can be merged into the current master branch.



### **Issues**

Throughout the development process, we plan to use the Git build-in feature of ‘Issues’ that allows a user to create an issue based on current code, or missing code. Every developer can see, follow and contribute to open issues by creating a branch and open a ‘pull request’ to be reviewed by other developers, and eventually merged into the master branch.

## Resources

### **Hardware**

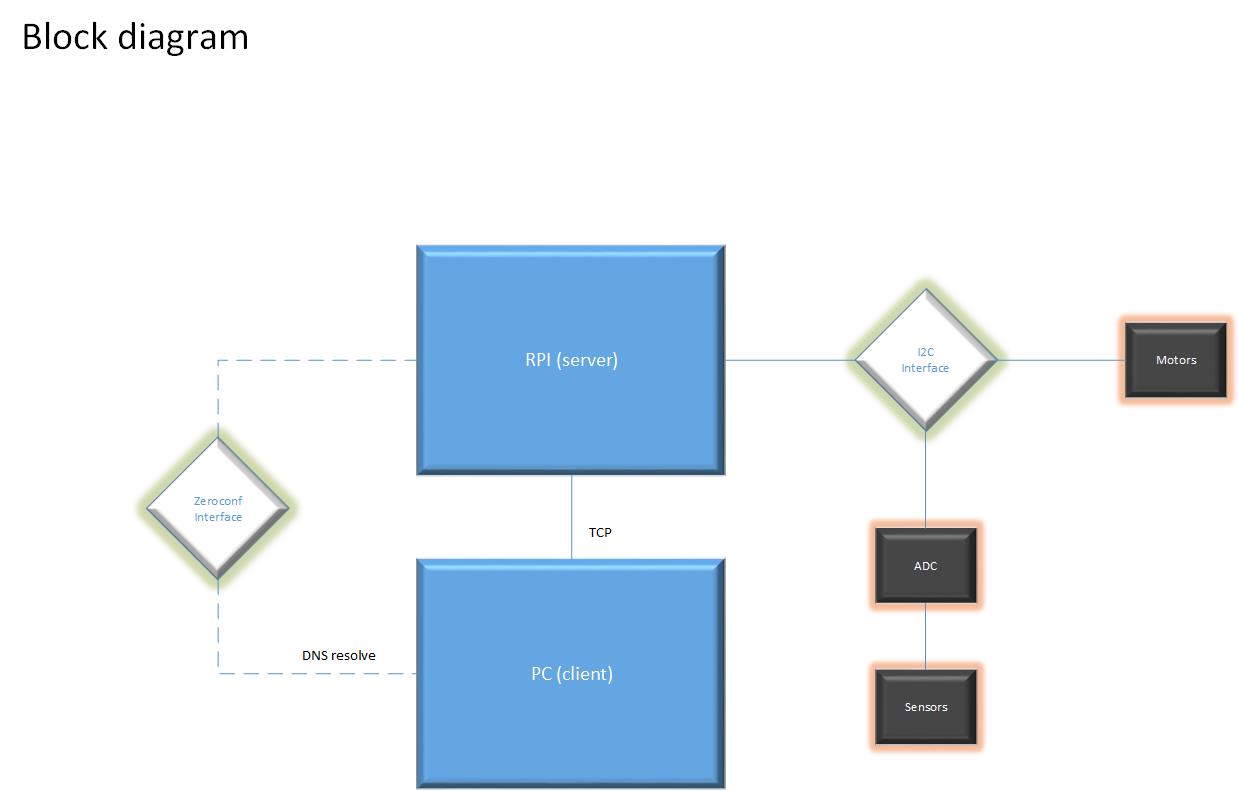
* Robot platform with DC motors
* Robot platform with Stepper motors
* Motor controllers for DC motor
* Motor controllers for stepper motor
* Ultrasonic sensors
* IR sensors
* AD7998
* Mice
* Raspberry PI´s
* WIFI USB adapter
* I2C expansion board
* Voltage regulator
* Battery pack
* Laptops
* Router

### **Software**

* Eclipse
* Github
* Dropbox
* Google drive

# Problem solution

Figure XX will show an overall block diagram



The robot involved two main functions: the ability to map the maze and the ability to over and over go through the maze to a selected target using the shortest path.

We saw the need for a state machine to handle which action we were in the midst of performing with an extra state where we could wait for instructions.

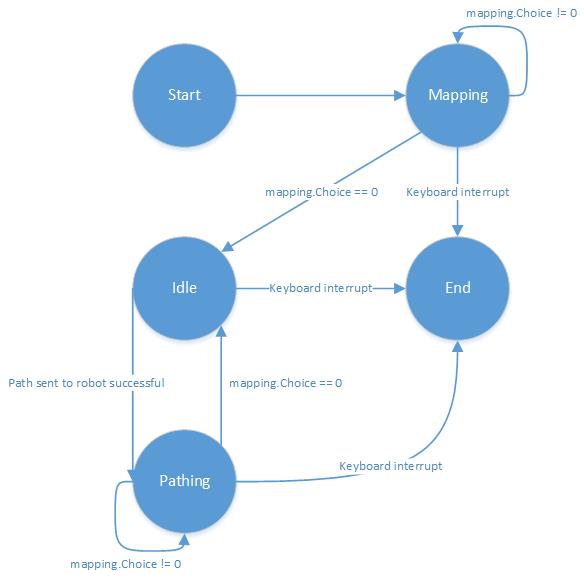
We divided the robot into three states:

1. Idle
2. Mapping
3. Pathing

In idle the robot would be able to receive requests and respond with data.

In Mapping the robot would traverse the maze and build a data structure representing the maze for future transmission to the PC.

In “Pathing” the robot would go through a path received from the PC and stop at the target cell. Figure XX will show a state machine diagram



We wanted to run the two main functions in threads which would give us the ability to check if a thread was alive and evaluate on that instead of using critical zones.

We designed the threads so when they had finished their tasks they would self-terminate and in the end be joined together with the main process.

## **Navigation**

### **Design**

### **Implementation**

### **TEST**

## **Mapping**

### **Design**

When faced with the objective to map a maze it was needed to generalize the specifications of a maze. Our maze was made of square cells side-by-side in a 2d coordinate system.

The square cells are divided either by an opening or a wall.

In 2.2.2 it was specified that the exam maze could held loops and dead-ends but not “islands” defining cells with walls in all directions.

We tried to outline the actions a robot would go through in order to traverse the maze.

The robot would need to turn, drive straight, make U-turns in dead-ends and have some sort of memory in order to avoid getting stuck in loops in the maze.

#### Position

In order to map a maze the robot needed to be able to reposition itself inside the rows and columns of the maze as well as a way of detecting whether a cell has been previously examined.

To reposition itself the robot needed a way of knowing the range it had traversed in a given direction. We decided to use steps as our approach towards odometry. This would work since our PID algorithm would help keep the robot in the middle of the path and thereby the counted steps traversed would come near the straight line between to points in a row or column.

#### Coordinate system

In order to explore the maze and to remember where we had been, a data structure for the underlying coordinate system was needed. Inside the data structure would be a table with cells, each cell occupied by a description of the walls surrounding it.

When we had settled on a way of position the robot and a data structure to hold the cells of the maze, we moved our focus to a procedure of exploring the maze.

#### Mapping algorithm

We started at the blackboard by drawing a maze similar to the current layout of the test maze present in the classroom and tried to identify movements to successfully complete the mapping. We saw that a maze consisted of four scenarios:

1. Corridors  
   The robot changes position in either rows or columns
2. Corners  
   The robot changes position in both rows and columns
3. Crossroads  
   The robot has to make a choice between different possibilities
4. Dead-ends  
   The robot can only make a U-turn

Of the four scenarios we started with the first. Since we already had auto-correction and a way of counting steps implemented, we decided to only make mapping actions in the other three scenarios.

Scenario 2 and 4 had a similarity in that only two possibilities existed in both: “stop” or “continue”. The difference was the option of “continuing”.

“Continue” in scenario 2 meant “turn” however in scenario 4, it meant “U-turn”

Since differences were minor these scenarios were classified as similar.

Scenario 3 stood out as representing the points where the path would branch out in different directions.

We decided to use “depth first search” to explore these branches. This algorithm normally suffers from infinite loops if faced with a loop however since we had a strong coordinate system with memory capabilities underneath this would not be a problem.

We specified our priority for the algorithm as following:

1. Straight
2. Right
3. Left
4. U-turn

Since scenario 3 was defined a situation with lack of auto-correction we felt that the priority of decisions could be neglected since both outcomes would result in insecurities regarding actual position and thereby accumulating error until next scenario 1.

#### Robot / PC - strategy

When the robot had traversed the maze and build up a maze model it would need a way of being transported to the PC application.

Since we already had a working WIFI connection to the robot we decided the PC should be able to request the maze over network when the robot had finished mapping the maze.

We then looked at the different possibilities of transporting our maze model data in a way that were standardized in both ends. Python offered pickles but since we had not decided the language of the PC application yet we decided to use JSON objects.

At the PC we explored the capabilities of Python doing the GUI and found that the QT library was sufficient in providing an easy “model-view-controller” patterned approach.

Briefly we had thought about using Java for the development of the PC application but using the combination of Python and QT kept us in a mono-linguistic development phase which eased our focus.

An application was developed in Python with a GUI and functionality for requesting mazes from the robot and displaying them. See Appendix XX for application screenshots of central functionality.

We wanted our robot and PC to communicate without too much setup so instead of having to configure options as IP in the network we wanted a more automatic approach to the infrastructure. We looked at making our own application protocol for establishing the TCP interactions but eventually a solution based on Zero-Configuration was implemented.

We wanted a simple communication platform so we decided to use “request-response” with the PC being the client and the robot being the server.

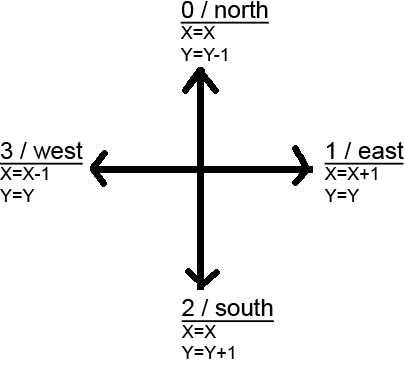
Looking at our state machine's three states we saw the risk of deadlocks since the robot simultaneously ran a TCP server taking requests for a map while the robot was building the map.

We solved this by running the “Mapping” state and “Pathing” state inside a thread. By doing this we could check and see if the thread was running before trying to obtain the map data. This solution was preferred because of simplicity compared to having a variable used to change the state inside a critical zone.

### **Implementation**

We started by creating the coordinate system and orientation concepts to move within.

In order to keep track of direction we decided to use cardinal directions as a global reference.

In FIGURE X underneath we show how we define the directions. 

By using these specifications we can also specify the rules of the cells in the maze.

A cell can be one of many situations that all can be generalized into a binary pattern using the defined directions.

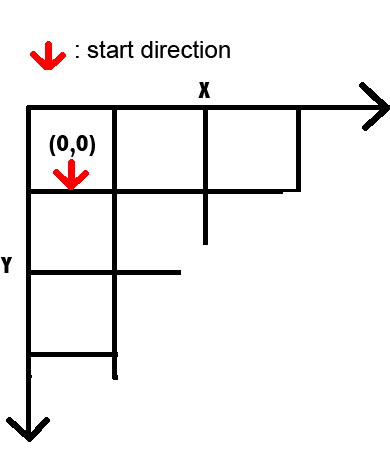
By using binary representations where 1 defines a wall and 0 an opening we can clockwise define a cell by the following list:

[north, east, south, west]

For example a cell with an opening north and west would be:

[0,1,1,0]

When placed in the maze the start direction of the robot is south and its start position is (0,0)

)

When mapping the robot goes through the flow described in Appendix XX

It starts by sampling the IR-sensors and parses the sample to a function that identifies which directions are blocked by a wall. The result of this evaluation is locally defined [left, right, straight] where 1 represents a wall and zero represents and opening.

We use this value to identify the different scenarios previously described.

When scenario 2 to 4 is experienced the walls list is passed to the mapping object that returns a locally defined choice with the exception that the mapping object also can return 0 recognized as “finished mapping”. If “0” the Mapping thread will set an event which will stop its while loop and in the end terminate itself.

The Mapping object will take care of making a local choice of direction based on current position and direction.

It does so by implementing a stack. Items are added to the stack at all corners, dead-ends and crossroads. At every iteration the Mapping object goes through the process described in Appendix XX

When the Mapping object adds to the maze table it writes an integer into the cells. The integer only has 4 bits used being the binary representation of the globally oriented walls at the current cell.

Example:

A cell with 4 walls north and east and openings south and west will be [1,1,0,0] and in the table it will be saved as an int with the same binary pattern: 0b1100 or 12.

Saving the walls list as an int simplifies the data structure and makes it easier to transfer to the PC as a JSON dictionary object.

In the Mapping object we used a simple procedure to translate local directions into global directions:

FIGURE X

For all the following it the result is wrapped to be within the boundaries of 0 to 3 according to the global directions IS IT REALLY ZERO TO THREE?

* Straight is current direction
* Left is current direction - 1
  + if < 0 then 3
* Right is current direction + 1
  + if > 3 then 0
* U-turn is current direction - 2
  + if < 0 then += 3

#### Robot / PC communication

When we implemented the TCP server at the robot we wanted it to be flexible and future proof.

Based on the chosen “request-response” model the PC application transmitted a JSON object with a key called “message”.

We chose to use “callback” functionality by implementing a dictionary inside the TCP server. in this dictionary you could add your own function and a message value it should react on.

Our TCP server receives and strips the JSON objects and looks for the key:“message”. it then takes the value from that key and tries to look it up in its dictionary of callbacks.

If it finds a callback it will run it with the value of the key:”params” also in the JSON object.

When the robot starts it registers its own callback functions at the tcp server. It registers functions to take care of sending maze models, receiving paths and sending current positions.

When asking for a maze our JSON object would look like this:

***{'message':"maze"}***

### **Test**

The mapping part was developed in a close cycle of testing and implementing.

We blackbox tested the mapping related functionality in two ways

1. Tests where the robot part could be simulated in a virtual machine
2. Tests that involved the robot in the actual maze

Using virtual environment we tested the parts involving communication between the robot and the PC. These blackbox tests can be seen in TESTCASE-1.

Using the actual robot we blackbox tested the ability to map a maze as seen in TESTCASE-3.

We also verified that the robot was able to write a logfile while driving.

During the process we used regressive testing every time we needed to fuse big parts into the main branch. Here we tested to see if new functionality had broken the software at placed previously been debugged or even new bugs that could have been created.

We whitebox tested our classes using main methods that were developed to test functionality. We used these to test the classes themselves every time code had been rewritten or reorganized. This process resembled unit-testing however in a more manual approach.

During the endphase we used destructive testing in order to reveal missed bugs etc. This proved a good idea since many parts had been left untouched while focus had been shifted during the process. Destructive testing revealed the loose ends otherwise forgotten.

From the tests we concluded that our mapping was functional and worked in the tested scenarios on mazes of different layouts and sizes. However our autocorrection from time to time served problems in such a way that in rare occasions we would leave a turn with such an angle that steps would be lost and the robot would lose its exact position in the coordinate system.

## **Path finding**

### **Design**

The robot had a main task of being able to go from a point to another point in the maze by following the fastest path.

Since our robot only used 90 degree turns, the fastest path would not only be lesser traversed cells but also lesser turns.

#### Defining the graph

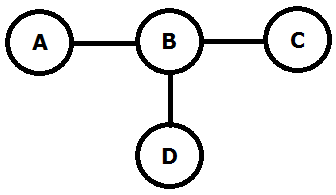
We started out early by planning to use flood fill algorithm. We wanted to use flood fill algorithm because it at the same time provided a way of traversing a maze creating a map as well as finding the fastest route between two points.

After sometime we realized that we couldn’t include the turn cost with the flood fill algorithm which we had to discard due to this and start finding a new algorithm for our path finding challenge. Another reason was also that we wanted to separate the process of mapping and path finding to the level of the robot doing the mapping and the PC doing the path finding.

Later in the process when the mapping process had spawned the specification of the maze model SEE REFERENCE we began researching the different ways of making path finding on the data structure in the maze model.

A critical point in most path finding algorithms is the use of a graph.

We focused on a way of translating the data of our maze model into usable basis for creating a graph.

For solving the problem of designing relations between each cell of the maze with a cost we looked into parent -> neighbor relation. In the illustration (FIGURE XX) the cost between the cells will be explained. In the case we will use cell A B C and D.

They are connected in the following way A to B (AB), B to C (BC), and D to B (DB). The cost between AB, and DB, will always be the same, in this example let’s call it 10. But the cost from BC will change depending on the previous node was AB or DB.

The reason for this cost different is, turns takes 3 times longer for the robot to execute then driving straight thru a cell. In this example if the robot is coming from BD the cost will change form 10 between BC to 30. This value will be used for the algorithm to find the fastest path.

The graph is implemented into a coordinate system where each note will have its own X, Y coordinate. To determinate whether the notes are a turn note or a drive straight note, we are looking on the change in X, Y coordinates. If only one of the coordinate’s changes, e.g. X to X1 and Y stays the same, we know that it a straight thru turn, but if both X and Y changes to X1 and Y1 we know it’s a turn and the cost will be adjusted. Using this data structure in theory look very easy to read and detecting errors.

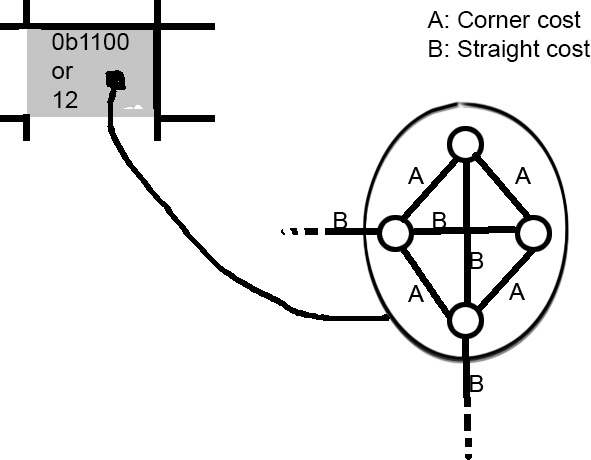
We started to investigate A-star which looked promising as a solution to the path finding issue since it had a heuristic function that would lower the search arena and thereby have a much faster time finding the path.

We started implementing A-star and it was able to operate on our graph using the turn cost process using parent-neighbor comparison but then we encountered a new error where we did not get the shortest path sometimes.

This error was unknown and still not explained why it happened. We tried to isolate the error by flipping off the heuristic function as well as trying different A-star implementations as well as increase the turn cost to 20 times the straight cost.

Since we were running late according to the milestone plan we decided to focus on creating a graph from scratch that already incorporated the turn cost differences.

After sometime we came up with the idea that creating 4 micro nodes (north, east, south, west connected as Figure XX shows) in every cell it would be possible to detect if we were turning and giving it an extra cost.



Every cell has at least one micro node that is connected to a neighboring cluster of micro nodes, the cost of the travel from those nodes would be a straight cost. The turn cost is only inside the cell with the micro cells within, north- south and east-west is the only ones with a straight cost but its bidirectional.

The next step after creating the new graph was to implement the Dijkstra algorithm to traverse it.

Our whole project has regarded the maze as undefined in size and thereby scalable. So we wanted to use Dijkstra to find specific paths and not pre-calculated all existing fastest paths from all locations to all locations. Thereby we chose to use CPU power over ram which makes sense since we run the path finding path on a PC.

#### Robot / pc strategy

We wanted the PC to transmit the path back to the robot over our established networking platform. At the robot we wanted to be able to receive a path with instructions and traverse the map using the instructions.

First we tried to send steps with each position in the final path. But this showed insufficient since the step counter often was imprecise. This would lead to premature instructions where the robot would execute a choice leading to wall since it thought it had reached the opening.

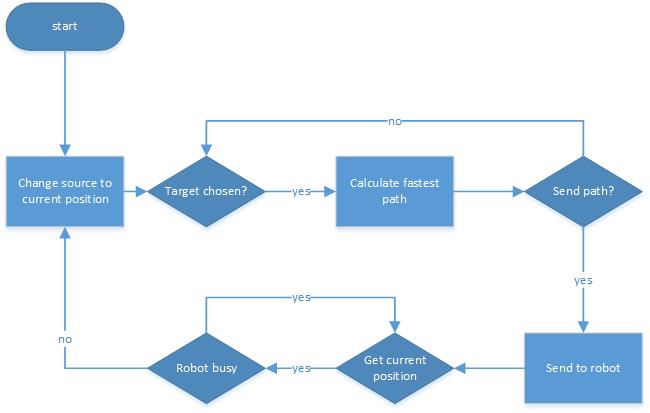
Our solution was a mix of the ability to react on scenarios where we broke off from driving in corridors together with the ability to drive to a certain position inside a corridor.

This solution meant that the robot would receive choices from the path at every dilemma (turn, dead-end or crossroad) and only in the last segment of the path the instructions would include steps. This would solve the problem of being able to drive to a point in the middle of a corridor where no dilemma existed.

To solve the issue with imprecise steps we decided that during the “Pathing” process the robot would traverse a known map and thereby loading its current position from the path would be good enough.

We made the path finding and pathing modules around the idea that the PC would acquire the current position of the robot and use this position as the start of the path.

In practice this meant a workflow described in following figure:



### **Implementation**

Graph:

Inside a structure of nested for loops we evaluated the content on an inspected cell in the maze model table.

The item in each cell was an int containing a bit pattern describing the walls surrounding it. So in order to interpret it we had to loop through all four cardinal directions. We checked for walls by using bit operators checking if 1 shifted in to the current inspected direction would match a with the content of the cell.

***for d in range(4):***

***tmp=1***

***if not (walls &(1<<(3-d))):***

***tmp=0***

***self.nodes[x][y][d]=Node(x,y,tmp,d,dillemma)***

***self.graph[self.nodes[x][y][d]]=[]***

Using the code above we created micro nodes for each existing direction containing a 0 if not facing a wall inside a cell.

Since our graph would be bidirectional the edges would be doubled.

When creating edges we used the following two approaches:

**For connections between micro nodes:**

*for i in all directions*

*for j in all directions*

*cost = straight cost*

*if absolute difference between i and j is not 2*

*cost = turn cost*

*append node i and j to eachother with cost*

**for connections between micro node clusters:**

*for all cells in table*

*if node(x,y) facing east and node(x+1,y) facing west exists with openings*

*append eachother to eachother in graph with straight cost*

*if node(x,y) facing south and node(x,y+1) facing north exists with openings*

*append eachother to eachother in graph with straight cost*

When the graph is created we use a standard implementation of Dijkstra to find the shortest path from source to target. see REFERENCE NEEDED (Wikipedia Dijkstra) for pseudocode.

### TEST

To test our path finding solution we started out by

# Conclusion

## Process assessment

Requirement traceability matrix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | Test cases | | | | |
|  |  | TC1 | TC2 | TC3 | TC4 | TC5 |
| Requirements | R1 |  |  | X |  |  |
| R2 | X |  |  |  |  |
| R3 | X |  |  |  |  |
| R4 |  |  |  | X |  |
| R5 |  |  |  | X |  |
| R6 |  |  |  |  | X |
| R7 |  |  | X |  |  |
| R8 |  | X |  |  |  |

# Bibliography

Internet sites

Datasheet:

<http://www.analog.com/en/analog-to-digital-converters/ad-converters/ad7998/products/product.html>

<http://www.sharpsma.com/webfm_send/1205>

<http://www.avrcard.com/Documents/datasheets/tmc222_datasheet_v105.pdf>

# Glossary

|  |  |
| --- | --- |
| PC Glossary | |
| PID controller | A proportional-integral-derivative controller |
| TCP | Transmission Control Protocol |
| DFS | Depth first search |
| IR | Inferred |
| Git | Github |
| RPi | Raspberry Pi |
| TC | Test Case |
| DHCP | Dynamic Host Configuration Protocol |
| DNS | Domain Name System |
| Zeroconf | Zero Configuration Networking |
| CD | Compact Disc |
| IP | Internet Protocol |
| LAN | Local Area Network |
| mDNS | multicast Domain Name System |
| DNS -SD | Domain Name System Service Discovery |
| GUI | Graphical user interface |
| JSON | JavaScript Object Notation |

# Appendix

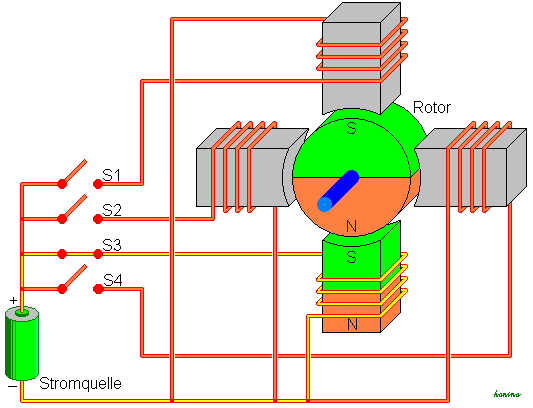
1. Milestone plan

missing

1. Theory

### **Motor**

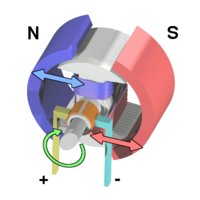
#### Stepper Motor

A stepper motor consists of a minimum of 4 iron cores. These iron cores are entangled in copper wire and by supplying electricity to these core a magnetic field is created. This is also called a Coil-magnet setup. In the middle of all the cores a shaft is placed. This shaft is magnetic and therefore has South and a North pole. When adding electricity to one of the coils in the shaft will align its North pole with the coils South pole. By alternating between which coil gets power the shaft will start tuning. By only tuning one coil on at the time the shaft will preformed a Full-Step rotation. This will make the shaft turn 90 degrees each time the next coil is activated. Turning on 2 coils, 1 coil, 2 coils, and 1 coil etc. the motor will generate a Haft-Step rotation and the shaft will turn 45 degrees on each impulse. This makes the increments much finer when the shaft is rotation and therefore more accurate. All these different combination of turning on coils is called StepModes.

To control this switching of electricity impulses to each coil, a motor controller is needed. The motor controller controls which coil needs power when, to perform the requested rotation.

<http://da.wikipedia.org/wiki/Step-motor>

#### DC Motor

A DC motor uses almost the same setup internally as a stepper motor, but the magnet-shaft and coils are switch and there are in most cases only 2 Coils. This means that the magnets are on the "outside" and the coil is on the "inside". (See the illustration). A DC motor receives direct current, and uses brushes located on the shaft to switch the polarities. The switching of poles happens internally inside the DC motor

and therefore in most cases a motor controller is not needed to make it work.

Due to the lack of a motor controller most DC motor, is more responsive then a stepper motor and also accelerate faster.

<http://en.wikipedia.org/wiki/Brushed_DC_electric_motor>

### Auto correction input

#### Odometry

Odometry is the collection of data from actuators, encoders or sensors, in order to estimate a position relative to the starting position.

#### Mouse

The idea of using mice odometry to estimate position is an idea we have had since the beginning of the project. Two mice are used to input relative coordinate-changes to our RPi whenever a displacement of the robot has happened. Several mathematical calculations are done based on the given x,y coordinate pair, and ultimately an length and angle of the movement is calculated. These two values are then used to attempt to estimate the change in position over time, and fed to a PID controller- calculation-algorithm to minimize heading errors, respectively**[[1]](#footnote-1)**.

#### Wheel Encoders

Another alternative to using mouse was the idea of using wheel encoders to determinate distance and direction. Setup on the robot is done by mounting gray-code Ref til et appendix med et gray code wheel to each wheel, with an sensor to read the bit-pattern. When the wheels are turning the sensors will recognize changes in the bit pattern and a position of the wheels can be determinate.

#### Sensors

The robot must be able to determine distances to the surrounding walls, in order to maintain a heading and know when to make a turn. Two primary options are given in this context; Infra-red sensors or Ultrasonic-sensors (or a combination of both). Either choice will work on a robot, but have different impacts on how the robot will function when running a maze. We have to look at the two sensor specifications and the robot behavior requirements, to make a satisfying engineerical choice

* **IR Sensor**

The infra-red sensor (which in our case is a GPD120x Sharp sensor) uses triangulation and CCD array to calculate a distance to objects. It works by emitting pulses of infra-red light, which in case of a present object, will be reflected and caught by the detector. This will create a triangle between the emitter, detector and point of reflection. The distance of the reflecting object will determine the angle in this triangle, which is then used by the CCD array to calculate a distance to the object.

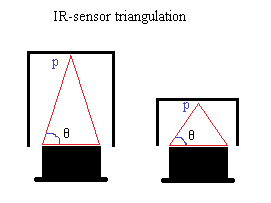


Figure 1: Shows the IR-sensor triangulation, where ‘p’ is the point of reflection and ‘theta’ is the angle created

The sensor output is non-linear with respect to the distance, so it is not insignificant where the sensors are mounted on the robot. Based on the graph of the analog voltage as a function of the distance, one will have to make decision of an optimal reading interval**[[2]](#footnote-2)**.

* **Ultrasonic sensor**

The ultrasonic sensor works differently from the IR-sensor, by emitting inaudible sound to detect surrounding objects, instead of light. We know that sound travels at a certain speed in air, so by keeping track of time elapsed since the sound was emitted and until the echo is detected, a very accurate distance can be calculated.

The sound emitted by the ultrasonic sensors spread out radially, which can cause problems, if several ultrasonic sensors are used, and not placed properly on the robot. If the sensors are pinging too close to each other, one sensor may read another sensors echo as its own. Furthermore materials and object angles might cause other problems such as lost echoes, or ghost echoes.

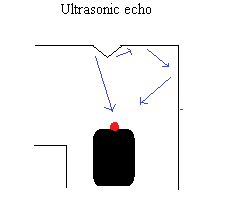


Figure 2: Shows ultrasonic sensor echoes. One echo reflects fine on the wall, the other turns into a ghost echo as a result of the angle on the wall

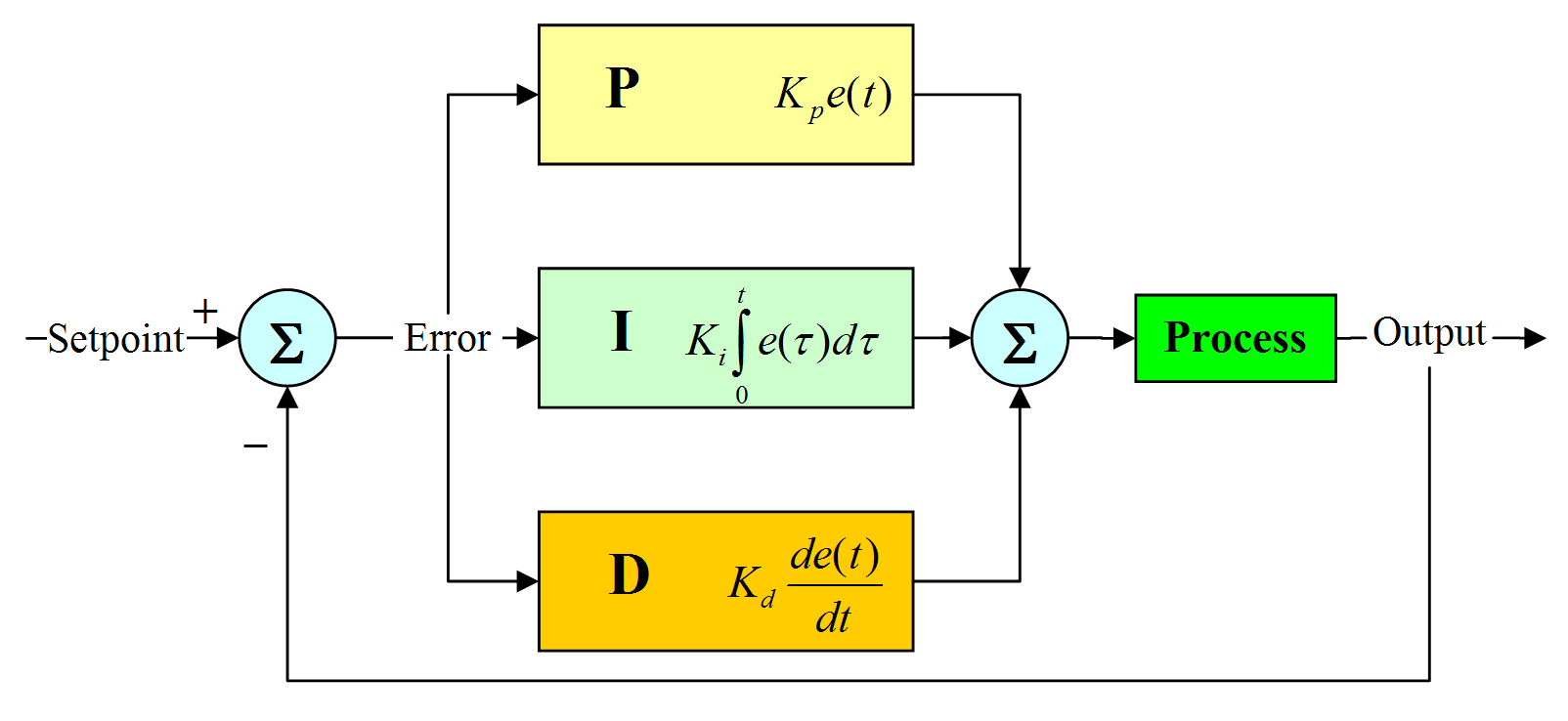
### Auto correction

#### PID

A Proportional – Integral – Derivative controller is a feedback controller, this means that it uses feedback response to determinate its output. A PID Controller calculates an error value in form of a difference from an input to a reference value.

Readouts from the Process block are added to the Setpoint value. From this sum an error will occur, it can be positive or negative. This error is past into the 3 blocks, P, I, and D block. All these values from the outputs are summed together after they have been multiplied with their individually gain factors Kp,Ki and Kd. The sum is then past into the Process. This loop is repeated until the Possess is terminated. (Look below for illustration)

|  |  |  |
| --- | --- | --- |
| P = Current error | I = The sum of previously errors | D = Estimate around future errors |



#### Iterative

The iterative solution consists of moving cell by cell and doing trigonometry calculations. The calculations is made when the robot have moved to its next target no calculations is made on ongoing movement. The robot moves from its current position to the middle of the next cell. First when it has reached its target the robot will first turn so it will be facing straight down its path, the degrees it needs to turn is calculated with help of the width of cell and the length of the distance read from the sensors + the distance from the sensors to the middle of the robot. Secondly the distance it moves each time is calculated with help of trigonometry by knowing the distance of a whole cell and the distance from half of the width – new distance from sensor + the distance from sensor to the middle of the robot. Finally it will find the degrees it needs to turn to be facing its next target; this is done by taking acos((half of the width – (new distance from sensor + the distance from sensor to the middle of the robot))/( the distance it needs to move)). There are two different scenarios depending on if the robots movement last going to the left or right.

### Path finding

Path finding is the process of finding a way between a source and a target. In computer science path finding is normally associated with finding the optimized path from A to B in terms of cost.

Path finding algorithms operate in a system of nodes connected by edges.

For easy understanding a node can be a station, an edge can be a transport route and the cost can be the time it takes to get from station A to station B. The system is described as a graph.

In such a system terminal cost from station A to station B can vary after the chosen path. You can choose to take route that goes near the harbour for fresh air but will end up with extra minutes used on the bus if the fastest route dictates not getting near the harbour.

In path finding the relationship between nodes is often described as parent <-> child.

In such a way the cost between a parent and a child describes the cost to move on in the process. Where the cost between a child and its parent can describe the memory of how much it cost to get to your present position.[[3]](#footnote-3)

A lot of different algorithms for path finding exist.

Important characteristics of algorithms can be divided in two.

1. Greedy:  
   Greedy algorithms that doesn't remember cost. They look ahead and make choices based on distance to target as well as cost to nodes.
2. Informed:  
   Informed algorithms which takes the cost until present position into account when making choices about future travel. On top of that they can have the characteristics of greedy algorithms with heuristics and cost to next coming nodes.

#### Dijkstra

Dijkstra finds the minimum cost to all nodes from its origin, which can be done for all nodes as its origin.

Dijkstra algorithm is often used for routing as it searches all paths.

If negative edges are used it's not possible to guaranteed the shortest path.

Dijkstra works by defining a source node and a target node. The source node sets a cost to 0 and all else nodes get a cost of infinity.

Dijkstra uses a priority queue to hold info on cost to nodes from source node.

The priority queue is prioritized with minimum cost as highest priority.

Dijkstra iterates through nodes starting with the source node and expanding with its neighbouring nodes.

While iterating it decides between investigating a neighbouring node or going back and investigating another branch defined in previous iterations.

Reconstruction of a path is based on each node having a pointer to its previous node.

At the initialization all nodes have their previous node set to null.

In this way a path can be reconstructed in the end by starting with the target node and iterating through previous nodes until previous node is set to null.

Dijkstra is mainly used for situations where the knowledge of cost to all nodes is needed. This is due to Dijkstra lacking direction in its search.

#### Flood fill

Flood fill can be used to scan an area and check for connections between nodes or for path finding. By flooding the area from a point (start node) to all other nodes you will be able to map the whole area. With path finding you can find each neighbour to the current node and mark it as example where you start will be node number 1 every node that can be connected to this node will be node number 2 then you will investigate node number 2 and mark all possible connections to those as number 3 and continue till you find them all then by numbering you will be able to find the cheapest path to your chosen target and use that path to follow

### Communication

#### Zeroconf

zeroconf or Zero-Configuration describes a process where the IP and port of an application can be registered and resolved automatically by nodes in a network.

In a network of nodes configured by DHCP you get an IP address but it doesn’t supply you with a way of resolving a specific service on a specific host, a task which is normally carried out by a DNS server.

Zero-Configuration solves this gap in a LAN by providing a service structure where servers can register a service and clients can browse for specific services.

Zero-Configuration is an umbrella-term describing many different technologies as mDNS, DNS-SD etc.

Major operating systems implement a library to take care of all these different approaches to resolve local services. Mac OSX uses “Bonjour”, Linux typically uses “Avahi”.

A typical Zero-Configuration setup would be a network printer. The printer would announce and register its service (printing) on a network by using a service name “printer”, a domain “.local” and a service type “\_ipp.\_tcp”.

Service types are defined by the DNS-SD organization and can be seen here:

<http://www.dns-sd.org/ServiceTypes.html>

A client on the network would then be able to use a Zero-Configuration browser either as a complete application or embedded inside an application, to check which services are available on the network. By doing so the client would see a service named “printer.local” with the service type “\_ipp.\_tcp” identifying a network printer.

### Mapping

#### Depth first search

Depth first search (DFS) is used for traversing or searching graph. it marks one node as its start/root and then it explores one of the branches all the way till it can’t go any further then it will start backtracking till it finds a unexplored branch and goes all the way to the end and will repeat this till it have explored every possibility the order which it choses it priority to search is chosen by the user of the DFS. You can go left right and straight in which ever you chose to priorities you set.

#### Graph

A graph refers to a set of nodes (cells) with edges (connection between nodes) connecting.

An undirected graph is a graph where there is no difference between the directions for the edges meaning that from node A to node B is the same, as from node B to node A.

A directed graph is a graph where a edges are not the same meaning that it got a direction from node A to node B, is not the same edge as from node B to node A and its possible to go one way only.

An edge can be weighted meaning it will have a cost (distance/time/price).

The most used ways to store a graph are adjacency matrix and adjacency list.

Adjacency matrix is a matrix that is representing which nodes of a graph are connected together with other nodes. The matrix is n x n where n = nodes. A matrix structure provides faster access but is consuming a lot of memory.

Adjacency list is a list of unordered collection of a graph one list for each node in the graph, each list shows a set of the nodes neighbours. List structures takes more time to access then a matrix but most cases uses less memory.

If the concern is memory then with a dense graph a matrix would be better than a list, but if it’s a sparse graph the list is better option as less memory is wasted.

1. Test cases

|  |  |
| --- | --- |
| **TC1: LOLCATS IN THE AIR** | |
| **Tested by:** | Team 3 |
| **Purpose:** |  |
| **Functional requirements:** |  |
| **Preconditions:** |  |
| **Test sequence:** |  |
| **Description of the expected result:** |  |
| **Result of the test:** |  |
| **Testers comments:** |  |

|  |  |
| --- | --- |
| **TC1: Receive and display a maze in a GUI** | |
| **Tested by:** | Team 3 |
| **Purpose:** | Verify that the PC can request and receive the maze map and display it graphically |
| **Functional requirements:** | R2, R3 |
| **Preconditions:** | The robot has mapped a maze and is in idle state.  The PC application is open and has identified and connected to the robot. |
| **Test sequence:** | 1. We click the get maze button 2. We wait for the robot to respond 3. The robot responds with a maze 4. The GUI opens a new window and displays the maze graphically. 5. The shown maze is compard with the aschii output from the robot. |
| **Description of the expected result:** | The PC is able to request and receive the map from the robot and displays a map identical to the one visible in aschii at the robot |
| **Result of the test:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:mazeReceivedTC1.pngNIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:mazeSolvdTerminalTC1.png  We checked the aschii output from the robot in the terminal and found that it was identical to the maze shown in the graphical window.  (Both figures are inverted to save ink) |
| **Testers comments:** | By looking at the terminal screenshot you can recognize the content of a maze cell describing its surrounding walls. |

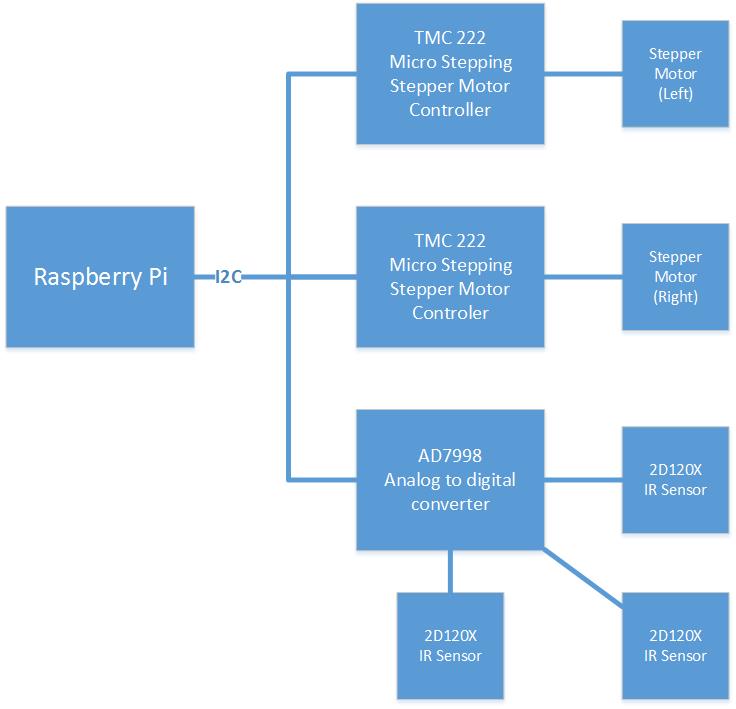
|  |  |
| --- | --- |
| **TC2: Autocorrection** | |
| **Tested by:** | Team 3 |
| **Purpose:** | See if the robot is able to correct itself in a corridor with an angled start direction |
| **Functional requirements:** | R8 |
| **Preconditions:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:robotPidStartingPosition.jpg  The robot is placed as the above figure shows. Facing angled towards the corridor.  The corridor is created with a length of 1.5 meters. |
| **Test sequence:** | 1. The robot is put into mapping mode 2. The robot will use PID autocorrection to align in the middle of the hall. 3. The program will be stopped by keyboard interruption. |
| **Description of the expected result:** | The robot has aligned center in the corridor with few overshoots. |
| **Result of the test:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:PID_KURVER_PG1_IG0_DG0.2.png  The robot aligned center after 4 overshoots over 10 seconds with overshoots maxing out at 5cm from the center of the corridor. |
| **Testers comments:** | None |

|  |  |
| --- | --- |
| **TC3: Map a maze** | |
| **Tested by:** | Team 3 |
| **Purpose:** | Verify that the robot is able to map a maze |
| **Functional requirements:** | R1, R7 |
| **Preconditions:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:robotStartPosition.jpg  The robot is able to autocorrect and is placed in in the maze at the position above facing south. |
| **Test sequence:** | 1. The robot is put into mapping mode 2. The robot will set out from its start position and map the maze exploring first the longest path and then backtracing and exploring sidepaths. 3. The robot will in the end output its maze in aschii in the terminal 4. The maze is verified according to the physical layout of the maze |
| **Description of the expected result:** | We expected the robot to first go straight and only turn if straight was not an option and start explore sidepaths when faced with a u-turn. |
| **Result of the test:** | We saw the robot exploring the maze according to our expectations and in the end it gave us a maze representation fitting with the pysical layout of the maze. |
| **Testers comments:** | In some tries autocorrect made the robot leave a sidepath in an angled fashion making it hard for it to identify travelled distance, however result stayed true to the physical layout. |

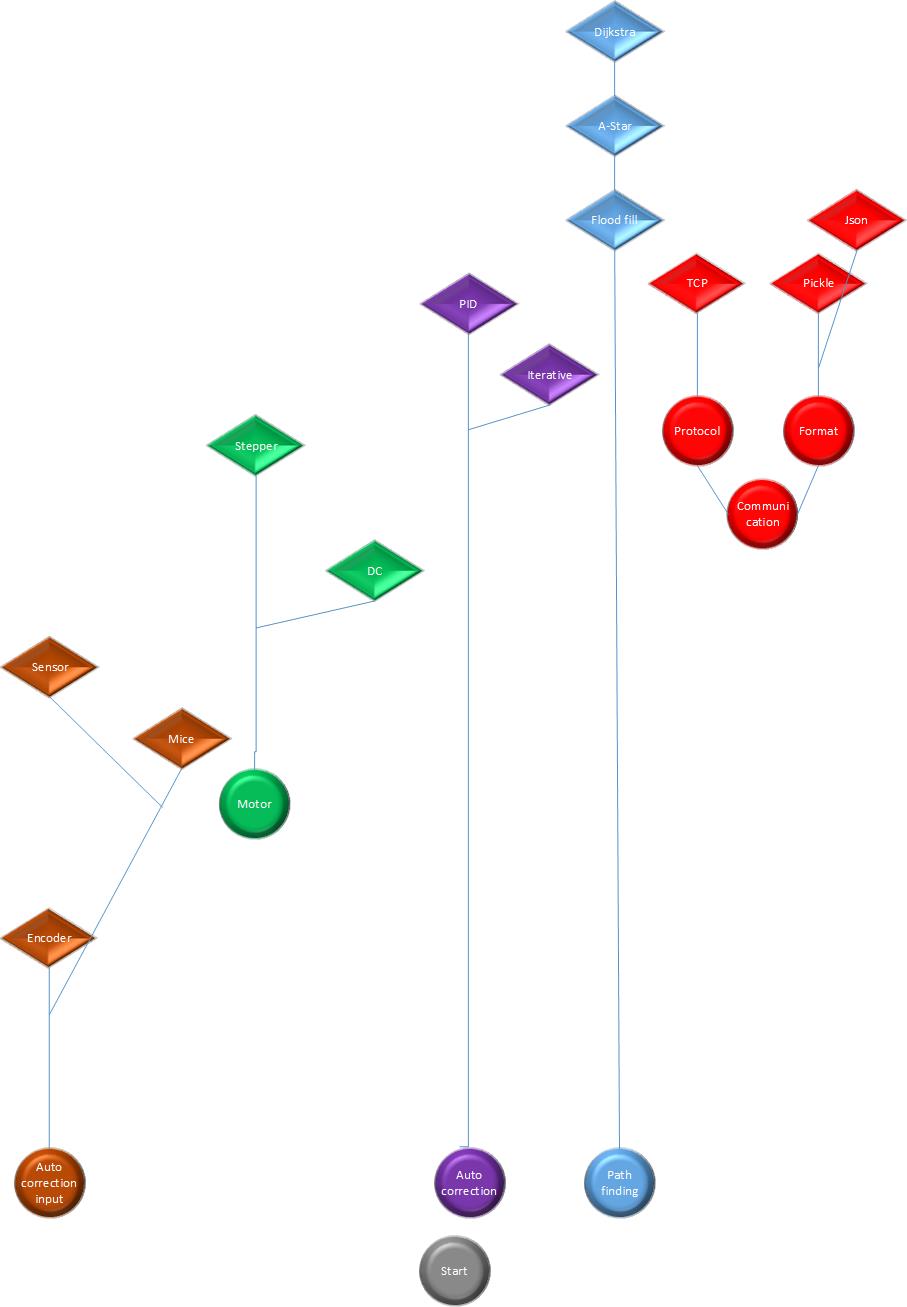
|  |  |
| --- | --- |
| **TC4: Fastest path** | |
| **Tested by:** | Team 3 |
| **Purpose:** | Verify that the fastest path algorithm finds the fastest path with fewest turns. And that the PC is able to send the maze. |
| **Functional requirements:** | R4, R5 |
| **Preconditions:** | Either a robot or a fictional test server is running a tcp server able to send a maze and respond with ”received, ok” when given a path over network.  The PC application is opened and waiting for a robot to register its zeroconf service. |
| **Test sequence:** | 1. We started a virtual server that responded with a known maze structure. The server was running in a virtual machine. 2. The gui application pops up and asks if we want to connect to the robot. 3. We press the get maze button 4. The server recognizes the request and responds with a maze model using JSON NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:fastest path:terminalSendPath.png 5. The GUI opens a view containing a graphical representation of the maze NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:fastest path:guiReceivedPath.png 6. We click the ”select and make path” button an chooses a target destination 7. We verify that it is the shortest path and send it to the virtual robot server 8. The Server receives the path |
| **Description of the expected result:** | We expect that our fastest path algorithm will find the fastest path will less turns and be able to successfully send this path to the robot |
| **Result of the test:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:fastest path:terminalReceivedPath.pngNIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:fastest path:calculatedFastestPath.png  We verified that the PC application was able to calculate the fastest path using a given target point and we could see using the colored circles scattered in the maze which cells the search algorithm had inspected.  We also verified that the virtual server was able to receive the path and reply that transmission was succesfull. |
| **Testers comments:** | The colored dots in the fastest path GUI figure shows which cells the algorithm had inspected colored after when the cell was inspected. Red = early, green = late. |

|  |  |
| --- | --- |
| **TC5: Drive a path** | |
| **Tested by:** | Team 3 |
| **Purpose:** | To comfirm that the robot recives the right path and follow the instructions correctly towards the target. |
| **Functional requirements:** | R6 |
| **Preconditions:** | That an user have sent a path |
| **Test sequence:** | 1. The robot have successfully received the path 2. The robot drives the received path using the instructions within. 3. When the robot have no instructions left it goes to idle state. 4. We request the current position from the robot and verify it as our chosen target position. |
| **Description of the expected result:** | We expect the robot to be able to receive the path and drive to its target destination. |
| **Result of the test:** | The robot drove until the target destination and reported a correct position back equal to our chosen target destination. |
| **Testers comments:** | none |

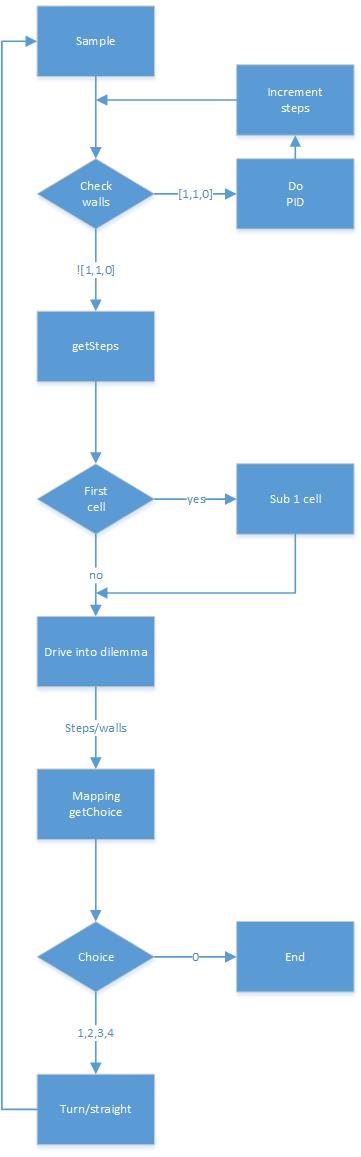
1. Hardware diagram



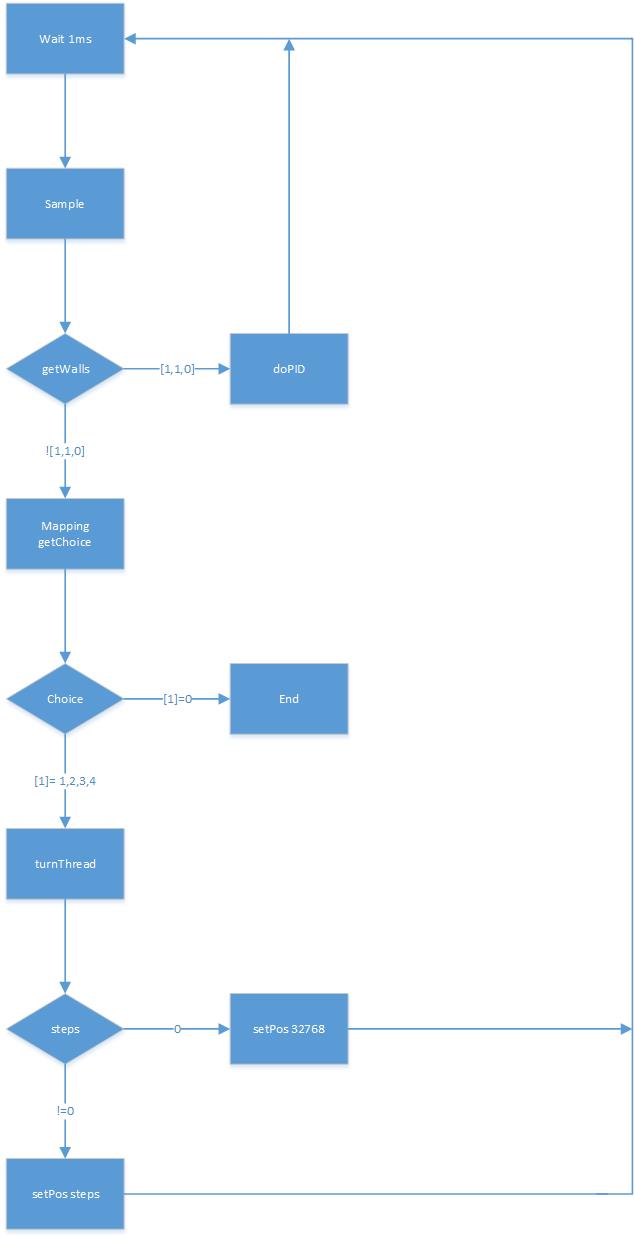
1. Process diagram



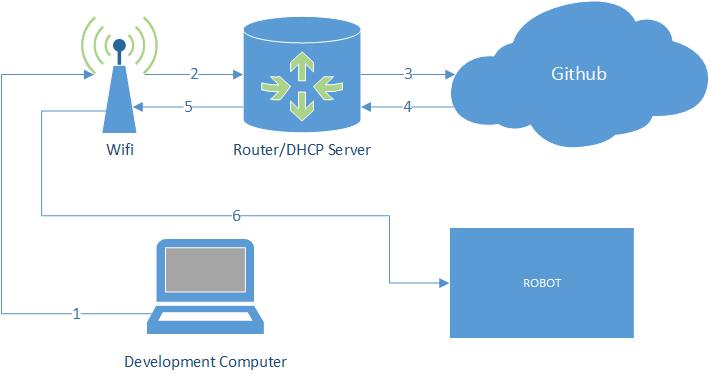
1. Mapping logic flowchart



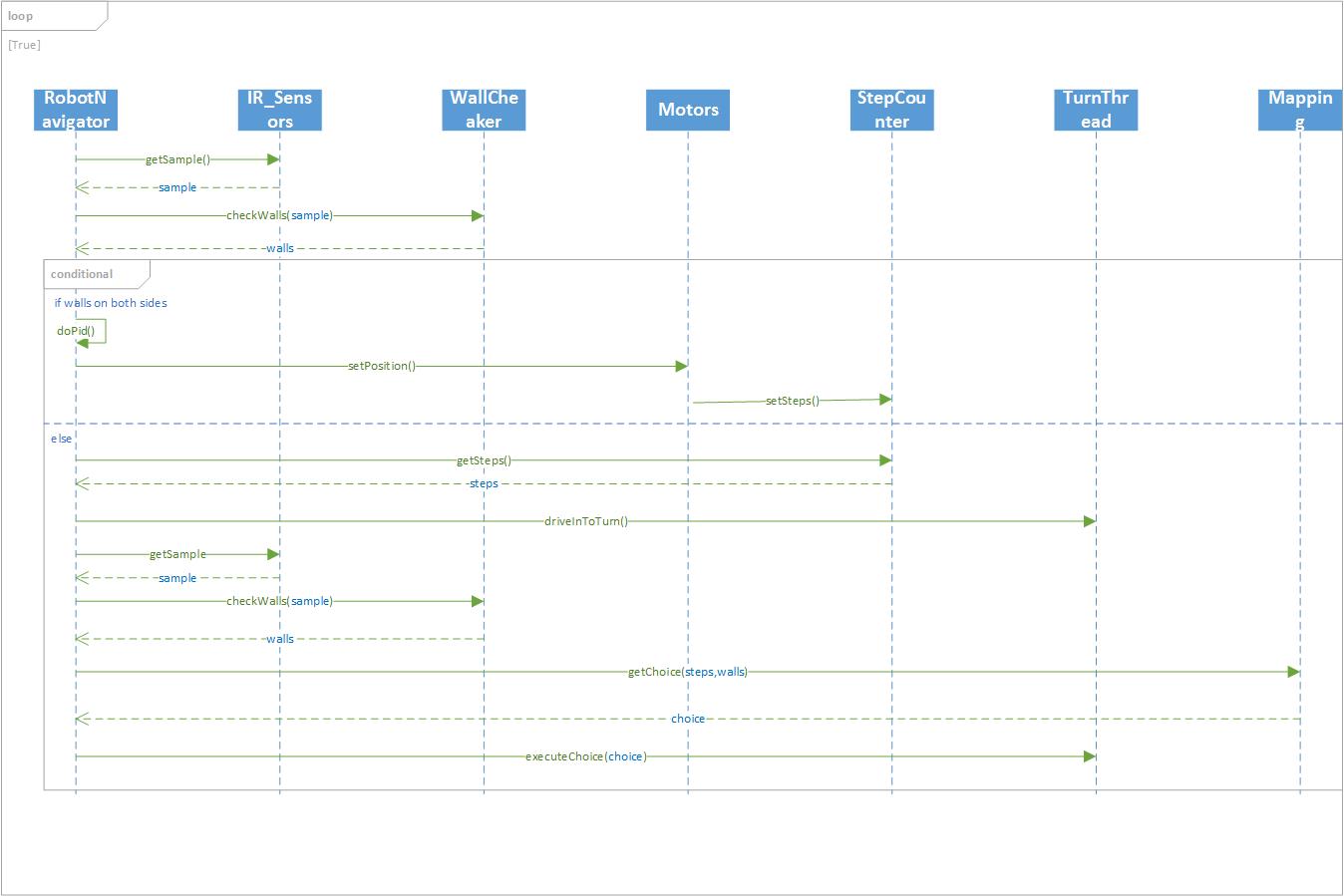
1. Mapping flowchart I dunno



1. Workflow diagram



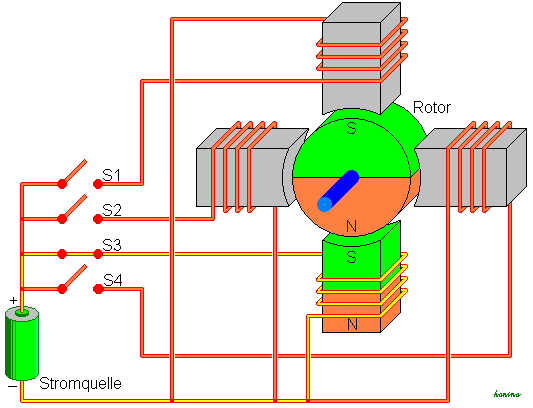
1. Sequence diagram





### **Motor**

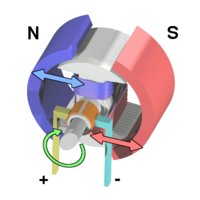
#### Stepper Motor

A stepper motor consists of a minimum of 4 iron cores. These iron cores are entangled in copper wire and by supplying electricity to these core a magnetic field is created. This is also called a Coil-magnet setup. In the middle of all the cores a shaft is placed. This shaft is magnetic and therefore has South and a North pole. When adding electricity to one of the coils in the shaft will align its North pole with the coils South pole. By alternating between which coil gets power the shaft will start tuning. By only tuning one coil on at the time the shaft will preformed a Full-Step rotation. This will make the shaft turn 90 degrees each time the next coil is activated. Turning on 2 coils, 1 coil, 2 coils, and 1 coil etc. the motor will generate a Haft-Step rotation and the shaft will turn 45 degrees on each impulse. This makes the increments much finer when the shaft is rotation and therefore more accurate. All these different combination of turning on coils is called StepModes.

To control this switching of electricity impulses to each coil, a motor controller is needed. The motor controller controls which coil needs power when, to perform the requested rotation.

<http://da.wikipedia.org/wiki/Step-motor>

#### DC Motor

A DC motor uses almost the same setup internally as a stepper motor, but the magnet-shaft and coils are switch and there are in most cases only 2 Coils. This means that the magnets are on the "outside" and the coil is on the "inside". (See the illustration). A DC motor receives direct current, and uses brushes located on the shaft to switch the polarities. The switching of poles happens internally inside the DC motor

and therefore in most cases a motor controller is not needed to make it work.

Due to the lack of a motor controller most DC motor, is more responsive then a stepper motor and also accelerate faster.

<http://en.wikipedia.org/wiki/Brushed_DC_electric_motor>

### Auto correction input

#### Odometry

Odometry is the collection of data from actuators, encoders or sensors, in order to estimate a position relative to the starting position.

#### Mouse

The idea of using mice odometry to estimate position is an idea we have had since the beginning of the project. Two mice are used to input relative coordinate-changes to our RPi whenever a displacement of the robot has happened. Several mathematical calculations are done based on the given x,y coordinate pair, and ultimately an length and angle of the movement is calculated. These two values are then used to attempt to estimate the change in position over time, and fed to a PID controller- calculation-algorithm to minimize heading errors, respectively**[[4]](#footnote-4)**.

#### Wheel Encoders

Another alternative to using mouse was the idea of using wheel encoders to determinate distance and direction. Setup on the robot is done by mounting gray-code Ref til et appendix med et gray code wheel to each wheel, with an sensor to read the bit-pattern. When the wheels are turning the sensors will recognize changes in the bit pattern and a position of the wheels can be determinate.

#### Sensors

The robot must be able to determine distances to the surrounding walls, in order to maintain a heading and know when to make a turn. Two primary options are given in this context; Infra-red sensors or Ultrasonic-sensors (or a combination of both). Either choice will work on a robot, but have different impacts on how the robot will function when running a maze. We have to look at the two sensor specifications and the robot behavior requirements, to make a satisfying engineerical choice

* **IR Sensor**

The infra-red sensor (which in our case is a GPD120x Sharp sensor) uses triangulation and CCD array to calculate a distance to objects. It works by emitting pulses of infra-red light, which in case of a present object, will be reflected and caught by the detector. This will create a triangle between the emitter, detector and point of reflection. The distance of the reflecting object will determine the angle in this triangle, which is then used by the CCD array to calculate a distance to the object.

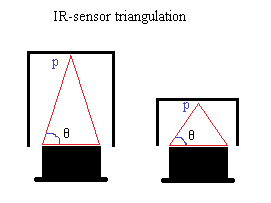


Figure 1: Shows the IR-sensor triangulation, where ‘p’ is the point of reflection and ‘theta’ is the angle created

The sensor output is non-linear with respect to the distance, so it is not insignificant where the sensors are mounted on the robot. Based on the graph of the analog voltage as a function of the distance, one will have to make decision of an optimal reading interval**[[5]](#footnote-5)**.

* **Ultrasonic sensor**

The ultrasonic sensor works differently from the IR-sensor, by emitting inaudible sound to detect surrounding objects, instead of light. We know that sound travels at a certain speed in air, so by keeping track of time elapsed since the sound was emitted and until the echo is detected, a very accurate distance can be calculated.

The sound emitted by the ultrasonic sensors spread out radially, which can cause problems, if several ultrasonic sensors are used, and not placed properly on the robot. If the sensors are pinging too close to each other, one sensor may read another sensors echo as its own. Furthermore materials and object angles might cause other problems such as lost echoes, or ghost echoes.

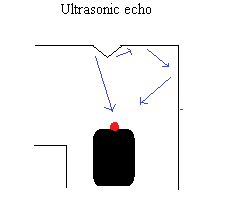


Figure 2: Shows ultrasonic sensor echoes. One echo reflects fine on the wall, the other turns into a ghost echo as a result of the angle on the wall

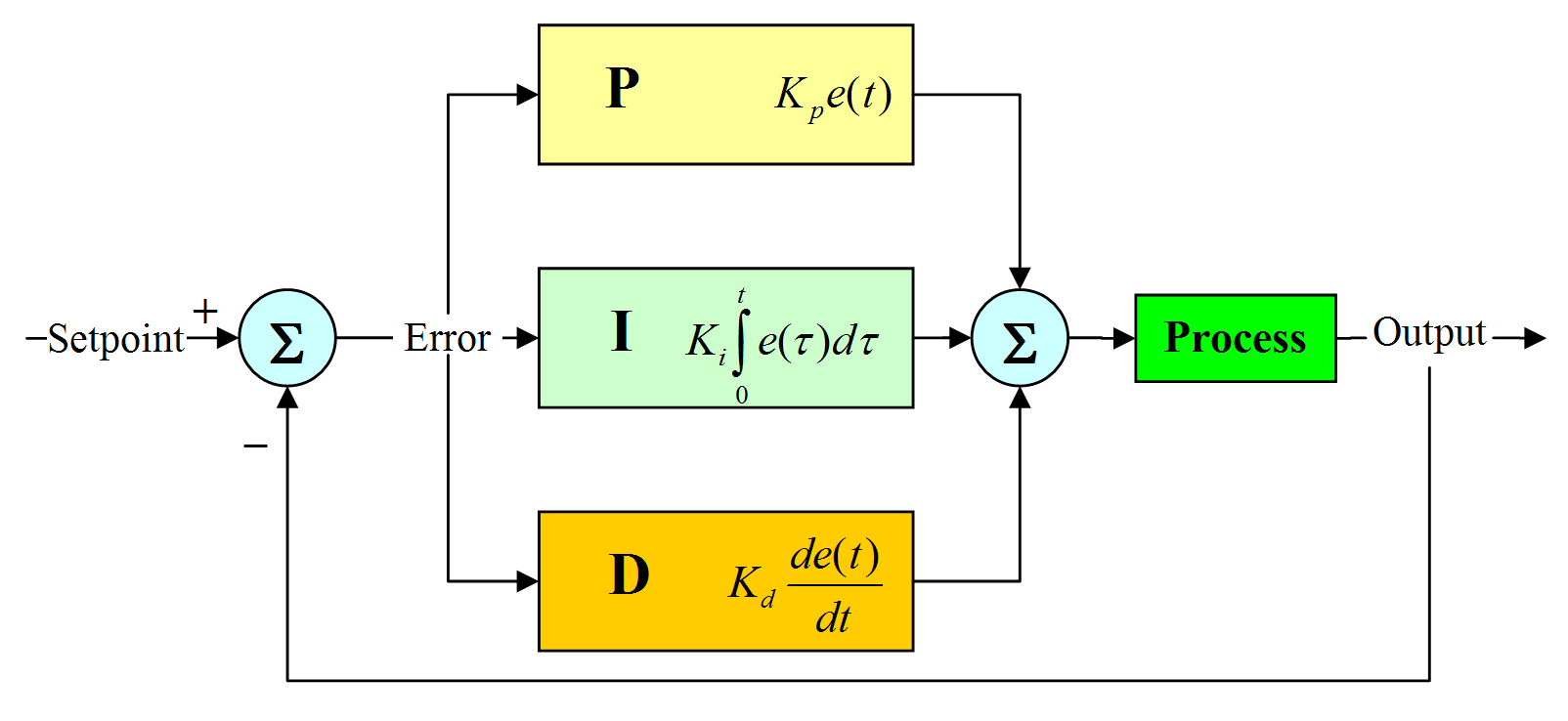
### Auto correction

#### PID

A Proportional – Integral – Derivative controller is a feedback controller, this means that it uses feedback response to determinate its output. A PID Controller calculates an error value in form of a difference from an input to a reference value.

Readouts from the Process block are added to the Setpoint value. From this sum an error will occur, it can be positive or negative. This error is past into the 3 blocks, P, I, and D block. All these values from the outputs are summed together after they have been multiplied with their individually gain factors Kp,Ki and Kd. The sum is then past into the Process. This loop is repeated until the Possess is terminated. (Look below for illustration)

|  |  |  |
| --- | --- | --- |
| P = Current error | I = The sum of previously errors | D = Estimate around future errors |



#### Iterative

The iterative solution consists of moving cell by cell and doing trigonometry calculations. The calculations is made when the robot have moved to its next target no calculations is made on ongoing movement. The robot moves from its current position to the middle of the next cell. First when it has reached its target the robot will first turn so it will be facing straight down its path, the degrees it needs to turn is calculated with help of the width of cell and the length of the distance read from the sensors + the distance from the sensors to the middle of the robot. Secondly the distance it moves each time is calculated with help of trigonometry by knowing the distance of a whole cell and the distance from half of the width – new distance from sensor + the distance from sensor to the middle of the robot. Finally it will find the degrees it needs to turn to be facing its next target; this is done by taking acos((half of the width – (new distance from sensor + the distance from sensor to the middle of the robot))/( the distance it needs to move)). There are two different scenarios depending on if the robots movement last going to the left or right.

### Path finding

Path finding is the process of finding a way between a source and a target. In computer science path finding is normally associated with finding the optimized path from A to B in terms of cost.

Path finding algorithms operate in a system of nodes connected by edges.

For easy understanding a node can be a station, an edge can be a transport route and the cost can be the time it takes to get from station A to station B. The system is described as a graph.

In such a system terminal cost from station A to station B can vary after the chosen path. You can choose to take route that goes near the harbour for fresh air but will end up with extra minutes used on the bus if the fastest route dictates not getting near the harbour.

In path finding the relationship between nodes is often described as parent <-> child.

In such a way the cost between a parent and a child describes the cost to move on in the process. Where the cost between a child and its parent can describe the memory of how much it cost to get to your present position.[[6]](#footnote-6)

A lot of different algorithms for path finding exist.

Important characteristics of algorithms can be divided in two.

1. Greedy:  
   Greedy algorithms that doesn't remember cost. They look ahead and make choices based on distance to target as well as cost to nodes.
2. Informed:  
   Informed algorithms which takes the cost until present position into account when making choices about future travel. On top of that they can have the characteristics of greedy algorithms with heuristics and cost to next coming nodes.

#### Dijkstra

Dijkstra finds the minimum cost to all nodes from its origin, which can be done for all nodes as its origin.

Dijkstra algorithm is often used for routing as it searches all paths.

If negative edges are used it's not possible to guaranteed the shortest path.

Dijkstra works by defining a source node and a target node. The source node sets a cost to 0 and all else nodes get a cost of infinity.

Dijkstra uses a priority queue to hold info on cost to nodes from source node.

The priority queue is prioritized with minimum cost as highest priority.

Dijkstra iterates through nodes starting with the source node and expanding with its neighbouring nodes.

While iterating it decides between investigating a neighbouring node or going back and investigating another branch defined in previous iterations.

Reconstruction of a path is based on each node having a pointer to its previous node.

At the initialization all nodes have their previous node set to null.

In this way a path can be reconstructed in the end by starting with the target node and iterating through previous nodes until previous node is set to null.

Dijkstra is mainly used for situations where the knowledge of cost to all nodes is needed. This is due to Dijkstra lacking direction in its search.

#### Flood fill

Flood fill can be used to scan an area and check for connections between nodes or for path finding. By flooding the area from a point (start node) to all other nodes you will be able to map the whole area. With path finding you can find each neighbour to the current node and mark it as example where you start will be node number 1 every node that can be connected to this node will be node number 2 then you will investigate node number 2 and mark all possible connections to those as number 3 and continue till you find them all then by numbering you will be able to find the cheapest path to your chosen target and use that path to follow

### Communication

#### Zeroconf

zeroconf or Zero-Configuration describes a process where the IP and port of an application can be registered and resolved automatically by nodes in a network.

In a network of nodes configured by DHCP you get an IP address but it doesn’t supply you with a way of resolving a specific service on a specific host, a task which is normally carried out by a DNS server.

Zero-Configuration solves this gap in a LAN by providing a service structure where servers can register a service and clients can browse for specific services.

Zero-Configuration is an umbrella-term describing many different technologies as mDNS, DNS-SD etc.

Major operating systems implement a library to take care of all these different approaches to resolve local services. Mac OSX uses “Bonjour”, Linux typically uses “Avahi”.

A typical Zero-Configuration setup would be a network printer. The printer would announce and register its service (printing) on a network by using a service name “printer”, a domain “.local” and a service type “\_ipp.\_tcp”.

Service types are defined by the DNS-SD organization and can be seen here:

<http://www.dns-sd.org/ServiceTypes.html>

A client on the network would then be able to use a Zero-Configuration browser either as a complete application or embedded inside an application, to check which services are available on the network. By doing so the client would see a service named “printer.local” with the service type “\_ipp.\_tcp” identifying a network printer.

### Mapping

#### Depth first search

Depth first search (DFS) is used for traversing or searching graph. it marks one node as its start/root and then it explores one of the branches all the way till it can’t go any further then it will start backtracking till it finds a unexplored branch and goes all the way to the end and will repeat this till it have explored every possibility the order which it choses it priority to search is chosen by the user of the DFS. You can go left right and straight in which ever you chose to priorities you set.

#### Graph

A graph refers to a set of nodes (cells) with edges (connection between nodes) connecting.

An undirected graph is a graph where there is no difference between the directions for the edges meaning that from node A to node B is the same, as from node B to node A.

A directed graph is a graph where a edges are not the same meaning that it got a direction from node A to node B, is not the same edge as from node B to node A and its possible to go one way only.

An edge can be weighted meaning it will have a cost (distance/time/price).

The most used ways to store a graph are adjacency matrix and adjacency list.

Adjacency matrix is a matrix that is representing which nodes of a graph are connected together with other nodes. The matrix is n x n where n = nodes. A matrix structure provides faster access but is consuming a lot of memory.

Adjacency list is a list of unordered collection of a graph one list for each node in the graph, each list shows a set of the nodes neighbours. List structures takes more time to access then a matrix but most cases uses less memory.

If the concern is memory then with a dense graph a matrix would be better than a list, but if it’s a sparse graph the list is better option as less memory is wasted.

# RPi-TMC222 baud rate Documentation

In order to maintain a continuous movement with the robot, we need to make sure the actual position of the robot, never reaches the target position. This is accomplished by reading certain registers in the TMC222 chip. Our getFullStatus2() method provides us with this information, in the form of:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Address | ActPos. | TagPos. | SecPos. | SecPos. | NA |

During our testing of getting this data, we have discovered huge variations of the information we receive, as well as complete useless return data. We suspected that this could be the result of wrong/inefficient baud rate of the I2C bus.

We have conducted three tests that show 40 status readings of the TMC222 chip, while the robot moves towards a target position. In each reading, the I2C baud rate is set differently. Also in each reading, the target position is set to 32000 micro-steps, followed by 25 intervals of 2500 micro-steps. Readings are done in a 1 second interval.

**First test: Baud rate = 100000**

**Expectation: We would expect ActPos to increase from 0 to 32000, and then increasing by 2500 until it overflows the TagPos limit of  and starts over. Also Address, SecPost, and NA should always stay the same (0xE0, 0x00, 0xFC and 0xFF).**

**Results:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Address | ActPos. | TagPos. | SecPos. | SecPos. | NA |

0xe0 32896 64896 0x80 0xfe 0xff

0xe0 34996 48895 0xff 0xff 0xff

0xe0 37859 64896 0x80 0xfc 0xff

0xe0 41094 64896 0x80 0xfc 0xff

0xe0 44453 64896 0x80 0xff 0xff

0xe0 48318 64896 0x80 0xff 0xff

0xe0 50565 64896 0x80 0xfc 0xff

0xe0 53171 64896 0x80 0xfe 0xff

0xe0 56534 64896 0xff 0xff 0xff

0xe0 59893 64896 0x80 0xfc 0xff

0xe0 63630 64896 0xff 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 64896 64896 0x80 0xfc 0xff

0xe0 64896 64896 0x80 0xfc 0xff

0xe0 64896 64896 0x80 0xfe 0xff

0xe0 64896 64896 0x80 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 37000 37000 0x80 0xfe 0xff

0xe0 39628 39628 0x80 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 44500 44500 0x80 0xfc 0xff

0xe0 47000 56319 0xff 0xff 0xff

0xe0 49628 49628 0x80 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 54500 54500 0x80 0xfc 0xff

0xe0 57000 57000 0x80 0xff 0xff

0xe0 59628 59628 0x80 0xfe 0xff

0xe0 62128 62128 0x80 0xfc 0xff

0xe0 64500 64500 0x80 0xfe 0xff

0xe0 34232 34268 0xff 0xff 0xff

0xe0 36860 36860 0x80 0xfc 0xff

0xe0 39360 39360 0x80 0xfc 0xff

0xe0 41860 41860 0x80 0xfe 0xff

0xe0 44232 38655 0xff 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 49360 49360 0x80 0xfc 0xff

0xe0 51860 51860 0x80 0xfe 0xff

0xe0 54232 43519 0xff 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 59360 59360 0x80 0xfe 0xff

**The test shows that all received data is complete rubbish and in no way useable to achieve a continuous movement.**

**Second test: Baud rate = 375000**

**Expectation: We would expect ActPos to increase from 0 to 32000, and then increasing by 2500 until it overflows the TagPos limit of  and starts over. Also Address, SecPost, and NA should always stay the same (0xE0, 0x00, 0xFC and 0xFF).**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Address | ActPos. | TagPos. | SecPos. | SecPos. | NA |

0xe0 0 32000 0x0 0xfc 0xff

0xe0 2222 32000 0x0 0xfc 0xff

0xe0 4951 32000 0x0 0xfc 0xff

0xe0 8179 32000 0x0 0xfc 0xff

0xe0 11656 32000 0x0 0xfc 0xff

0xe0 15384 32000 0x0 0xfc 0xff

0xe0 17622 32000 0x0 0xfc 0xff

0xe0 20350 32000 0x0 0xfc 0xff

0xe0 23578 32000 0x0 0xfc 0xff

0xe0 27056 32000 0x0 0xfc 0xff

0xe0 30783 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 34500 34500 0x0 0xfc 0xff

0xe0 37000 37000 0x0 0xfc 0xff

0xe0 39500 39500 0x0 0xfc 0xff

0xe0 42000 42000 0x0 0xfc 0xff

0xe0 44500 44500 0x0 0xfc 0xff

0xe0 47000 47000 0x0 0xfc 0xff

0xe0 49500 49500 0x0 0xfc 0xff

0xe0 52000 52000 0x0 0xfc 0xff

0xe0 54500 54500 0x0 0xfc 0xff

0xe0 57000 57000 0x0 0xfc 0xff

0xe0 59500 59500 0x0 0xfc 0xff

0xe0 62000 62000 0x0 0xfc 0xff

0xe0 64500 64500 0x0 0xfc 0xff **Overflow!**

0xe0 1464 1464 0x0 0xfc 0xff

0xe0 3964 3964 0x0 0xfc 0xff

0xe0 6464 6464 0x0 0xfc 0xff

0xe0 8964 8964 0x0 0xfc 0xff

0xe0 11464 11464 0x0 0xfc 0xff

0xe0 13964 13964 0x0 0xfc 0xff

0xe0 16464 16464 0x0 0xfc 0xff

0xe0 18964 18964 0x0 0xfc 0xff

0xe0 21464 21464 0x0 0xfc 0xff

0xe0 23964 23964 0x0 0xfc 0xff

0xe0 26464 26464 0x0 0xfc 0xff

**The test shows that with a baudrate of 375000, the robot behaves as expected, and returns correct information.**

In conclusion, we can derive that the serial communication between the robot and our raspberry pi, won’t work with a low baud rate. My idea is that with a low baud rate, we cannot read the data from the TMC222 registers fast enough, and as a result the register overflow and return corrupt information.

1. The complete mice odometry documentation is enclosed in the appendices [↑](#footnote-ref-1)
2. Full voltage output documentation is enclosed in the appendices [↑](#footnote-ref-2)
3. <http://cs.stackexchange.com/questions/553/how-do-common-pathfinding-algorithms-compare-to-human-process> [↑](#footnote-ref-3)
4. The complete mice odometry documentation is enclosed in the appendices [↑](#footnote-ref-4)
5. Full voltage output documentation is enclosed in the appendices [↑](#footnote-ref-5)
6. <http://cs.stackexchange.com/questions/553/how-do-common-pathfinding-algorithms-compare-to-human-process> [↑](#footnote-ref-6)