

Construction of a Ground-Based SLAM Capable Robot

Liam Brand

1. ABSTRACT

This report details the research, planning and development in order to create a ground-based autonomous robot capable of tracking its own location whilst mapping an environment.

DECLARATION

I haven't copied anything, honest. And nobody else wrote this stuff. Just me. And if I've even so much as glanced at somebody else's work, I've referenced it properly.

You can fail me if I'm telling lies.

2. ACKNOWLEDGEMENTS

My supervisor is an exceptional chap without whom this dissertation would have been possible. The University should bestow upon him immediately the title of Professor and adjust his remuneration accordingly.

Other people, too, have been quite helpful in getting me to this stage in my career.

CONTENTS

1. Abstract	2
2. Acknowledgements	4
3. Introduction	11
3.1 What is L ^A T _E X?	11
3.2 Why use L ^A T _E X?	11
Part I Analysis	12
4. Introduction	13
5. Mobile Robotics	14
5.1 Introduction	14
5.2 Movement	14
5.3 Rangefinding	16
5.3.1 Infrared	16
5.3.2 Ultrasonic	17
5.3.3 LIDAR	18
6. An Investigation Into SLAM	19
6.1 Introduction	19
6.2 What is SLAM?	19
6.3 Uses of SLAM in Industry	19
6.4 The SLAM Problem	20

Contents	6
6.4.1 Full SLAM	21
6.4.2 Online SLAM	21
6.4.3 SLAM Taxonomy	22
6.5 A Look At A Potential Solution	24
6.5.1 Introduction	24
6.5.2 CSM	24
6.5.3 Suitability of CSM	25
6.6 An Investigation Into SLAM - Conclusions	25
7. Product Requirements	27
7.1 Functional	27
7.2 Non-Functional	27
7.3 Security	28
8. Review of Tools and Techniques	29
8.1 Tools	29
8.1.1 Chassis	29
8.1.2 Microcontroller	31
8.1.3 Rangefinder	32
8.1.4 Software Framework	34
8.2 Techniques	35
8.2.1 Acquiring Scan Data	35
8.2.2 Dead Reckoning	37
9. Conclusions	39
 Part II Synthesis	 40
10.Introduction	41
11.Methodological Approach	42
11.0.1 Initial Prototype	43

Contents	7
12.Design	44
12.1 Introduction	44
12.2 Initial Overview	44
12.2.1 Hardware	44
12.2.2 Software	46
12.3 Designing for Requirements	46
12.3.1 Movement	46
12.3.2 Observation	48
12.3.3 Processing Observational Data	52
12.4 Final Design Documentation	54
12.4.1 Class Diagram	54
12.5 State Machine	54
13.Implementation	56
13.1 Introduction	56
13.2 Initial Setup	56
13.2.1 Hardware	56
13.2.2 Software	56
13.3 Implementing Requirements	57
13.3.1 Movement	57
13.3.2 Observation	58
13.3.3 Processing Observational Data	62
14.Testing	66
14.1 Introduction	66
14.2 Integration Testing	66
14.2.1 Basic Movement Testing	67
14.2.2 LIDAR Testing	67
14.2.3 File Writing Tests	68
14.2.4 Mapping Tests	68
14.3 System Testing	69
14.3.1 Test Results	69

Contents	8
Part III Evaluation	71
15.Introduction	72
16.Product Evaluation	73
16.0.1 Functional Requirements	73
16.0.2 Non-Functional Requirements	74
16.0.3 Conclusions	75
17.Process Evaluation	76
17.1 Objectives	76
17.2 Skills	77
17.3 Project Plan	78
17.4 Choice of Equipment and Techniques	81
18.Conclusions and Recommendations	83
18.1 Conclusions	83
Part IV Appendices	88
19.Terms of Reference	89
20.Testing	98
20.1 Tests	98
20.1.1 Integration Tests	98
20.1.2 Movement Tests	98
20.1.3 LIDAR Tests	98
20.1.4 System Tests	98
20.2 Test Code	103
20.2.1 Basic Movement Testing	103
20.2.2 LIDAR Testing	103
20.2.3 SDFFileSystem Testing	104

LIST OF FIGURES

5.1	Generated path ignoring the robot's non-holonomic properties	15
5.2	Wheels with 90 and 45 degree rollers	16
6.1	The SLAM problem illustrated [15]	21
8.1	4WD Omni-Directional Robot Chassis	29
8.2	3WD Omni-Directional Robot Chassis	30
8.3	Arduino Uno Rev3	31
8.4	FRDM-K22F	31
8.5	FRDM-K64F	32
8.6	RPLIDAR A2M8	33
8.7	RPLIDAR A1M8	34
11.1	Prototyping Methodology	42
12.1	Chassis assembly guide	45
12.2	Example shield setup	47
12.3	Shield connection	47
12.4	LIDAR Underside	49
12.5	LIDAR Underside	49
12.6	Scan Request	50
12.7	Response Descriptor	50
12.8	Scan Response Breakdown	51
12.9	Class Diagram	55
12.10	Finite State Machine	55

LIST OF TABLES

8.1	Time taken to write 16000 dummy samples to the Micro SD-Card	36
12.1	Components and respective voltages	45
12.2	RPLIDAR SDK files	51
13.1	LIDAR Microcontroller pin setup	59
13.2	SDFFileSystem Setup	63
14.1	Directions that each number represents	67
20.1	Movement Integration Tests	99
20.2	LIDAR Integration Tests	100
20.3	File Writing Integration Tests	100
20.4	File Writing Integration Tests	101
20.5	System Tests	102

3. INTRODUCTION

The human desire for exploration and the thirst to understand the world around us is as old as humanity itself. No matter the point in history, humans have sought out and explored the unknown. Carthaginians explored the coast of Africa. Romans, utterly terrified of what they might find, bravely ventured to Great Britain to seek out whatever might be there. Most famously explorers like Christopher Columbus set out and discovered the New World. As time has passed and the humankind's understanding of their world has been developed, we as a species have turned our attention outwards into the vast universe of which we are an unimaginably minute part of.

Part I

ANALYSIS

4. INTRODUCTION

The aim of this project is to develop a mobile robot capable of utilizing Simultaneous Localization and Mapping, often referred to within robotics as SLAM. SLAM concerns the ability for a robot to move around whilst simultaneously tracking its position and generating a map of the environment it navigates around. This will entail elements of mobile robotics, with regards to the appropriate selection of hardware followed by the assembly of said hardware to create the robot and the development of the software that will run on the mobile platform. Alongside this will be the development of a solution to the SLAM problem to allow the mobile platform to map the environments that it navigates around.

In order to implement this system and solve the presented problems, a clear understanding of the relevant fields needs to be established. The first aim of the analysis is to obtain just that. To do it, there will be a critical review of relevant literature. This will allow a good foundational base to build on, and will allow a greater understanding of how the problems presented by the project are to be solved. From this, solutions as to how these problems will be solved will be ascertained. Following this appropriate tools for the product will be identified, both hardware and software.

5. MOBILE ROBOTICS

5.1 Introduction

The creation of a ground-based mobile robot will require an understanding of the fundamentals of mobile robotics. The aim of this chapter is to develop such an understanding, looking over the chosen literature and discussing what can be learned for it as well as the steps that would need to be taken for the mobile platform to be fully realised.

5.2 Movement

This section aims to explore how movement can be achieved in the world of mobile robotics. From this, a greater understanding of robotic movement should be achieved which will aid the robot's development.

At its most basic level, movement in mobile robotics consists of determining a path between the robot's current position and its destination. Satisfying any movement tasks in this fashion will involve breaking down movement tasks into motions that accommodate for different constraints, such as the robot's capability and obstacles in the environment. One such constraint that has a significant impact in determining this path is the holonomic (or lack thereof) properties of the robot.

Most conventional vehicles use standard wheels that have only two degrees of freedom. These wheels can either roll forwards or backwards. This means it is a non-holonomic vehicle, which essentially means at any given point in the vehicle's state there are certain directions it cannot travel in. Barraquand and Latombe[4], both experts in their field with Latombe publishing one of the most cited works in the field of mobile robotics, discuss nonholonomic robots and their navigational properties. The example they give to demonstrate non-holonomic constraints is a car. A car is capable of obtaining any position within an environment, but the car's wheels are only capable of moving forward or backward. Therefore, the car's achievable velocities at any given point are two dimensional. This is what makes it a non-holonomic vehicle. A common real world example of this being an issue is in parallel parking.

Barraquand and Latombe go on to discuss the problems that non-holonomic navigation possesses. In constructing a collision free path, the first difficulty is in actually determining a feasible path between two points. This is because a straightforward translation of the robot from its initial position to its destination is not always possible. Indeed, Laumond et al[18] in their discussion of the issues with non-holonomic navigation mention that a determined path doesn't always correspond to a trajectory that can be used by the robot. In an article for IEEE, Murray and Sastry[24] demonstrate this by comparing a path generated by a path planner to the actual path that would need to be taken to translate a non-holonomic robot to a destination, seen in fig 5.1.

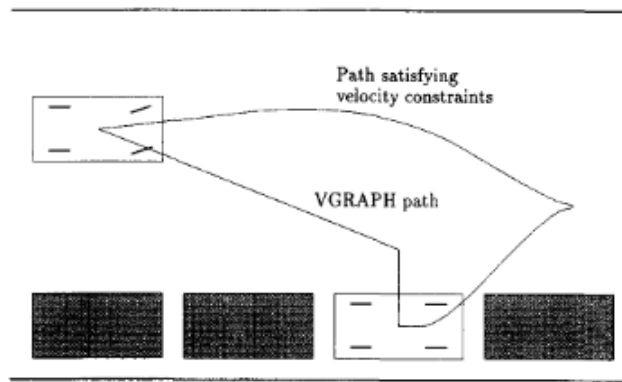


Fig. 5.1: Generated path ignoring the robot's non-holonomic properties

One of the ways in which these problems can be negated is through employing a holonomic vehicle, which would allow it to enjoy the ability to move in any direction regardless of its current position or heading. One of the most popular ways in which vehicles obtaining holonomic properties are constructed is through the use of special omnidirectional wheels.

Watanabe[39], a Professor at Okayama University's Faculty of Engineering who has published a great deal of work in robotics, discusses a few different variations of these omnidirectional wheels. One of the more popular variations in his discussion is the universal wheel, sometimes also referred to as the swedish wheel or mecanum wheel. This wheel is a larger wheel that has many rollers on the rim which allows the wheel to slide in a direction perpendicular to its motor axis.

The use of such wheels greatly simplifies the robot's movement. Calculated paths between points can be translated much easier into usable commands given the robot's movement freedom, and the number of manoeuvres required is reduced drastically since there is no need for repeated "backing up" movements to adjust the robot's heading.



Fig. 5.2: Wheels with 90 and 45 degree rollers

Watanabe also discusses ball wheels, with one well known implementation of such a wheel being ballbot by Lauwers et al[19]. The ballbot design involves a singular ball instead of any wheels, with legs that can deploy when the robot needs to be stationary. Whilst there are some potential stability issues during movement, with one experiment noting that the robot would lean in the direction of its destination as it traveled, it was still able to resist attempts to tip it over. Whilst interesting, in the context of the project the design of this is a bit too complex for it to be used for a mobile platform.

5.3 Rangefinding

Used for obstacle navigation and to assist in navigation, mobile robots often feature hardware that allows the robot to observe its surrounding environment. In the context of the project, this will also serve as a way to provide relevant data to allow for mapping.

This section aims to explore some of the more common rangefinding techniques by briefly explaining and evaluating them.

5.3.1 Infrared

Infrared (IR) sensors work by measuring things via the reflection of infrared light. The sensor will send out some infrared light where it will be reflected off of an obstacle. A receiver will capture this reflected light and depending on factors such as how much light is received back and the triangulation of how the light was received the presence of an obstacle will be determined.

In a paper produced for the International Journal of Advanced Robotic Systems regarding the usage of IR in mobile robotics, Do and Kim[13] elaborate

on some of the advantages of IR. First is their incredibly low cost, with a great number of IR sensors available online for very cheap prices. Websites like RobotShop and HobbyTronics list most of their sensors between £5 and £10. Additionally, IR sensors are able to achieve very narrow beam angle. This allows for precise detection, and can mean IR sensors excel where other sensors such as Sonar encounter environmental limitations such as spaces where wave reflection might cause inaccurate readings.

These benefits have been enough to see IR's usage robotics projects. Malik and Yu[22] for example utilised a ring of IR detectors for performing obstacle detection with an autonomous robot. Their implementation of it was quite successful, the robot demonstrated basic navigational abilities such as moving through doors and finding its way around unknown environments, successfully avoiding obstacles in the process. Dang and Suh[11] implemented a tracker using IR. In their experiments, their created devices succeeded in tracking humans and demonstrated an error (tracking inaccuracy) of less than 1.5cm when it was used for real world applications such as following supermarket carts.

IR has some downsides however. First, infrared sensors have both a maximum and a minimum range. Not only does the maximum range need to be factored in as sensor will struggle to pinpoint the location of light that has been reflected at a large range, but the sensor will also struggle to 'see' very close obstacles. Lee and Chongp[20] in their discussion of the different rangefinding techniques touch on IR's faults and mention that IR can have trouble with object detection depending on the colour of the object reflecting the IR light.

5.3.2 Ultrasonic

Ultrasonic sensors function by sending out ultrasonic pulses and measuring the amount of time it takes for these pulses to bounce back.

Like IR, Sonar sensors tout a relatively low price range towards the low end, with sites like HobbyTronics and RoboShop featuring sensors as cheap as £6. This would allow for a cheap way to implement coverage around the entirety of a robot, as multiple sensors could be attached and configured on the outside of the robot's chassis. Sonar's sound based nature means it isn't negatively affected by aspect such as heat or colour.

Sonar isn't without problems however. In an article for IEEE, Borenstein and Koren[7] discuss some of the drawbacks of ultrasonic range finders during their foray into using it for obstacle detection. The first issue was that detection could be very unreliable depending on the angle the sonar energy hit the surface at. Parallel surfaces were detected without issue but more

perpendicular surfaces caused a lot more of the sonar energy to be scattered and resulted in the detection being less reliable. This issue appears to get worse the smoother a surface is, with things like polished wood and plastics giving the worst results. Another problem was accuracy. Depending on the sensor's orientation and the angle at which an obstacle reflected the sonar energy, the received readings could be incredibly noisy. As well, sensors sending and receives pulses in close proximity to each other can have issues with interference. Whilst the article is 30 years old, the papers mentioned previously in the discussion of IR - despite being significantly more recent - still noted similar drawbacks to the use of sonar in their discussion of it[13][20]. We can infer from this that these issues are most likely due to the way in which sonar works, rather than the technological ability of the sensors used in the experiments.

5.3.3 LIDAR

LIDAR (Light Detection And Ranging) is a technology that uses light sensors to measure distances between the sensor and the target object. It achieves this by sending out light pulses which bounce off of objects back at the sensor where they are collected.

LIDAR's biggest strength is arguably the accuracy that can be achieved compared to other rangefinding techniques, which has led to its adoption for projects where accurate measurements are needed. One field that LIDAR enjoys much usage in because of this strength is forestry. Dubayah and Drake[14] mention the strengths of LIDAR for measuring forest features such as canopy heights, with similar praises for the accuracy are given by Lim et al[21] investigation into LIDAR's effectiveness for measuring forest structure. Some commercially available sensors boast very impressive statistics, with some ranges exceeding 10 metres whilst providing several thousand samples per second with 360 degree coverage [34]. Some of these sensors also feature SDKs (Source Development Kits), meaning the core sensor functionality will be accessible straight away allowing development to focus on the robot's logic rather than being bogged down in the belt and braces implementation of preliminary functionality.

All of these features come with an expected financial downside however. These features are expensive, with some sensors being as high as £350. Even lower end options with much smaller sensors that don't offer full 360 degree cover can cost up to £40.

6. AN INVESTIGATION INTO SLAM

6.1 Introduction

The development of a moving robot is only one half of the end product. As previously mentioned in the project's Terms of Reference the purpose of this project is also to develop a robot that is capable of self navigation and mapping. In order for this to be possible, the robot must be capable of using observations about its environment to build a map. Not only that, but it also must track its own location within this environment. This chapter aims to explore SLAM, a computing problem with research and implemented solutions that deal with exactly that.

6.2 What is SLAM?

SLAM stands for Simultaneous Localization and Mapping, and is something sometimes employed by mobile robots. Localization refers to the ability for the robot to be aware of its location within an environment, for example knowing where it is within a room. Mapping simply refers to building a map of the environment, such as the room the robot is in. SLAM is performing both of these tasks at the same time. Durrant-Whyte and Bailey[15], both academics that have done extensive work in the field of mobile robotics, best sum it up as the ability for a mobile robot to be placed at an unknown location in an unknown environment and then both create a consistent map of the environment and be able to accurately determine its location within this map. Similar definitions can also be found in other articles [9, 12].

6.3 Uses of SLAM in Industry

There are a myriad of potential uses for SLAM, many of which can be seen within the wider industry. Commercially it has been used for products such as vacuum cleaners, Dyson for instance has a small automated vacuum cleaner called the 360 Eye which employs SLAM techniques to map the areas that it moves around and cleans. SLAM has seen many uses in

archaeological contexts owing to its ability to perform exploration without risk to human life, one team [10] developed an underwater robot that used SLAM in order to map underwater cisterns that had been built thousands of years ago. The uses have not gone unnoticed by larger organisations. One of the research organisations within the USA's Department of Defense has held challenges (known as the DARPA Grand Challenge) offering cash prizes as incentives to create high value research. These challenges involve organisations submitting cars that are timed as they race around certain environments. NASA have also made use of it in the past, in 2007 they used an autonomous underwater robot [37] employing SLAM to go to the bottom of the world's deepest sinkhole. The robot used sensors to generate a sonar map of the sinkhole's inner dimensions 318 meters below the surface. The drone also tested technologies that could be used in other more extreme underwater environments such as the oceans under the crust of Europa, one of Jupiter's moons. This has led to increased interest being expressed in using SLAM for planetary rovers, which would allow for the mapping and navigation of different planet surfaces.

6.4 The SLAM Problem

Let's use some key notations to help break down the essentials of the SLAM problem.

\mathbf{t} - Current time.

\mathbf{x}_t - Location and orientation of vehicle.

\mathbf{u}_t - Control vector, for example drive forward 1 metre.

\mathbf{m}_t - True location of i th landmark within the environment.

\mathbf{z}_t - Observation of i th landmark taken at time t .

From these notations we can derive some sets.

$\mathbf{x}_{0:t} = \{ \mathbf{x}_{0:t-1}, \mathbf{x}_t \}$ - History of all vehicle locations.

$\mathbf{u}_{0:t} = \{ \mathbf{u}_{0:t-1}, \mathbf{u}_t \}$ - History of odometrical information pertaining to the robot's movement.

$\mathbf{m} = \{ \mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_n \}$ - Set of all landmarks.

$\mathbf{z}_{0:t} = \{ \mathbf{z}_{0:t-1}, \mathbf{z}_t \}$ - Set of all landmark observations.

Ultimately we want to use the robot's control inputs and observations to receive a map of the environment and the robot's path.

SLAM is generally approached probabilistically. This means that the attempted solutions factor in uncertainties within the data. Therefore, solutions to the SLAM problem will not act with exact certainties. For example, rather than saying the robot is in an exact location we would treat it as a

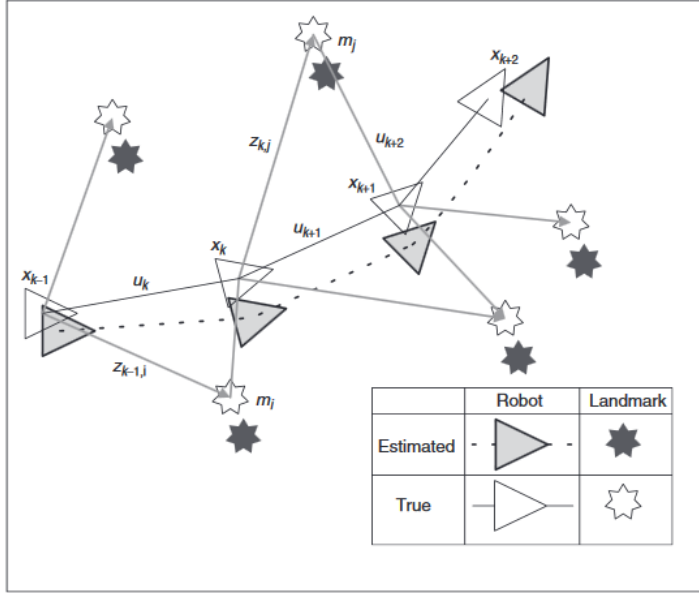


Fig. 6.1: The SLAM problem illustrated [15]

general location it is the most likely to be in. We want the probability distribution to be an estimation of current vehicle location and landmarks based on landmark observations and control inputs or odometrical data.

There are variations within the SLAM problem however. At a broader level, SLAM problems generally come in one of two flavours. These are full SLAM and online SLAM.

6.4.1 Full SLAM

Full SLAM involves using landmark observations and data relevant to discerning the robot's current position in order to determine the robot's entire path. It can be written as such -

$$p(\mathbf{X}_{0:t}, \mathbf{m} \mid \mathbf{Z}_{0:t}, \mathbf{U}_{0:t})$$

We want the probability distribution (essentially the estimation of the robot's current position) to be an estimation of what is on the left (current vehicle location and landmarks) based on what's on the right (landmark observations and control inputs).

6.4.2 Online SLAM

Online SLAM differs slightly in that it seeks to determine the robot's current location rather than the robot's entire path. It can be written as such -

$$p(\mathbf{x}_t, \mathbf{m} \mid \mathbf{Z}_{0:t}, \mathbf{U}_{0:t})$$

Compared to the previous notation used in 6.4.1, this probabilistic estimation only estimates the robot's position (\mathbf{x}) at the current time (t) rather than the robot's full historical path.

6.4.3 SLAM Taxonomy

The possible differences in the SLAM problem don't end there. Depending on different factors there are also different sub approaches to the SLAM problem. Below are some common variants.

Volumetric versus Feature-Based

Volumetric SLAM samples the map as a resolution high enough to allow a photo realistic reconstruction of the environment [36]. The map gained from this is generally high dimensional, but as the area increases in size and scale the map becomes significantly more complex. Feature-based SLAM simply extracts key features from measurements, with the map being solely made up of these features. This might be used if it is decided that only key features are of interest or if large parts of the mapped space are empty, as volumetric SLAM in these cases would be storing voxels that hold no geometric data of significant value [38]. As you would expect, this is quicker and more efficient but discards a lot more data than volumetric.

Topological versus Metric

Topological SLAM captures key places and their connectivity to other measured locations. Metric SLAM attempts to model the environment using geometrically accurate positioning. Metric SLAM would show the accurate positioning of various environmental features, topological would show them in relation to each other (e.g. place A is adjacent to place B) [36]. A good analogy would be a bus route map (topological) that displays the different stops versus showing the bus' actual route on a geographical map of the area (metric).

Known versus Unknown Correspondence

This entails relating the identity of sensed landmarks to other sensed landmarks. In known correspondence the identity of the landmarks is known, if a landmark is observed and then the robot moves and observed another landmark, the identity of the landmarks being known would let us to determine

if this landmark observation is the one we saw before or a newly observed one. Unknown correspondence would simply mean that in this situation we wouldn't know.

Static versus Dynamic

Static and Dynamic here refers to the environment. Static SLAM algorithms assume that no changes will take place in the environment whereas Dynamic SLAM methods allow for these changes to take place.

Small versus Large Uncertainty

The ability to represent uncertainty is another aspect. Some SLAM approaches will assume a very low uncertainty in the robot's location estimation. This might be when the robot is moving up and down a simple path, as it's much easier to guess where it's likely to be. Large amounts of uncertainty might occur however in more complex environments where locations can be reached from multiple different directions, or if the robot starts travelling in more complex paths that intersect with each other.

Activate versus Passive

Active SLAM involves the robot actively exploring its environment whilst it builds a map of it. Passive SLAM is when the SLAM algorithm is purely used for observation, with some other entity controls the robot's movement.

Single-Robot versus Multirobot

Single-robot simply refers to SLAM happening only on a single platform. Multirobot SLAM (sometimes known as cooperative SLAM) involves multiple robots often communicating with each other to merge their maps into a larger collective model.

There are multiple different paradigms that can be used to solve the SLAM problem, and each of these paradigms has many different implementations. One technique that has seen usage for solving the SLAM problem in autonomous mobile robotics is the Canonical Scan Matcher, generally referred to as CSM. This is the solution that we will be looking to implement for the project.

6.5 A Look At A Potential Solution

6.5.1 Introduction

As previously discussed there are a myriad of variations on the SLAM problem, and there are a few different paradigms used to implement solutions to it. During some preliminary research, one method of SLAMming came up that seemed like it would be suitable for the project called CSM.

6.5.2 CSM

CSM is an open-source C implementation of an ICP variant known as PIICP. It has seen usage for industrial prototypes of autonomous robotics, one of the most notable examples of this being Kuka, a German manufacturer of industrial robotics. It isn't quite a fully fledged SLAM solution, instead performing pairwise scan-matching on scan data that is fed into it. Before the PIICP algorithm it is based on can be explained, we must first look at the base ICP algorithm.

ICP

ICP stands for Iterative Closest Point, and it refers to an algorithm that attempts to minimize the difference between two clouds of points, something known as point matching or point set registration. In essence, it means getting one set of points aligned to another set of points. Besl and McKay present the algorithm as a statement [5] in their paper, and ICP is shown in terms of C++ in the Mobile Robot Planning Toolkit [6].

We first of all have a source map, and then we have a map that we wish to align to it which we will refer to as the reference map. We then go through each point in the source map and which point in the reference map is the closest to it. We then determine a transformation which would minimize the mean squared error (the average squared difference) between the two points before applying this transformation to the reference set and then going through this set of steps again. This is repeated until the mean squared error falls below a certain threshold.

PIICP

Censi explains PIICP in a series of steps [8]. To start with, we take a reference scan, a second scan and a first guess for the translation needed to try and match the two maps. We then generate a polyline of the reference map

by connecting sufficiently close enough dots (using a threshold). Following this, a loop similar to the one in the base ICP algorithm begins.

We first determine the coordinates of the second scan in the first scan's frame of reference using our initial translation guess. Then, for each point in the second scan, we determine the two closest points to it in the first scan. We trim any outliers within these matches, and use the sum of the squares of the distances from the points to the line containing the matches two points to find the error function. PIICP then uses an algorithm in order to minimize this error function which we now use as our translation guess. This new guess is used on the next iteration of the algorithm. This loop continues until either we have a convergence between the maps or a loop is detected as no further progress is being made.

6.5.3 Suitability of CSM

Firstly we can see that CSM is a pure C implementation of the previously described algorithm. This is excellent for the project, not just for the benefits of C such as it being a relatively quick language but also because it should be directly usable with an embedded board which would be the ideal choice for controlling the robot. Had it been in any other language we might have needed to use some sort of shared library to get it to work which likely would have slowed things down and potentially made the robot less effective.

CSM is however not a product of a professional company dealing in these matters, its open source nature could put some doubts with regards to its usefulness or reliability. However, it was developed by Dr Andrea Censi, someone who is a Deputy Director for the Chair of Dynamic Systems and Control at ETH Zurich meaning it is far from an amateur project. As previously mentioned as well it has been adopted by the German robotics company Kuka, so clearly it has enough merit to be used at the industrial level. Ultimately it would appear CSM is an ideal choice for the robot's localization and mapping functionality.

6.6 An Investigation Into SLAM - Conclusions

First and foremost we can safely establish that the SLAM problem is what we are addressing with regards to the implementation of the robot's ability to track its own location whilst mapping its environment. In addition to this we understand the fundamentals of the SLAM problem. This will massively benefit development, as being aware of having to store details internally such as wheel revolutions gained via odometric sensors will allow us to cut

down on time spent during development having to rewrite core pieces of functionality to make room for this.

7. PRODUCT REQUIREMENTS

Based on the different aspects of mobile robotics and SLAM that have been looked at as well as preliminary project objectives outlined in the Terms of Reference, we can outline the basic product requirements that will need to be achieved in the course of the product's development.

7.1 Functional

- The robot must be capable of movement
- The robot must be capable of observation
- The observational data must be processed into a map

7.2 Non-Functional

- **Build Quality** - The robot should be built to an acceptable standard. Wires should be managed to ensure the robot's function isn't impeded by them (e.g. getting stuck in the wheels) and components should stay inside the chassis during the robot's operation.
- **Functional Speed** - The observation and map generation should be done within an acceptable time. The robot shouldn't take more than 10 seconds to save readings, and it shouldn't take more than 30 seconds to generate a map from the readings once they have been acquired.
- **Adaptable Mapping** - To ensure the robot maps correctly, the mapping should be tested against a few different layouts. This will ensure the implemented method of mapping can successfully adapt to and map new environments.
- **Reliable Mapping** - When mapping the same environment multiple times, the robot's produced maps shouldn't differ too much. Repeated testing in the same environment will be carried out to ensure produced maps are consistent.

7.3 Security

There are no security requirements for the system.

8. REVIEW OF TOOLS AND TECHNIQUES

This chapter aims to focus on evaluating the potential tools and techniques that will be employed for the project's implementation. It will explore appropriate potential hardware and software, arriving at a conclusion along with an explanation as to why a certain tool or technique has been chosen.

8.1 Tools

8.1.1 Chassis

A few of the different chassis found during the preliminary research will now be evaluated to find which would be the most suitable for the project.

4WD 58mm Omni Wheel Arduino Robot - £260.78

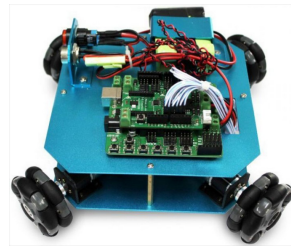


Fig. 8.1: 4WD Omni-Directional Robot Chassis

This chassis features four of the aforementioned omnidirectional wheels along with appropriate motors. The motors have encoders with them which record wheel revolutions, allowing us to use odometric measurements should we connect these encoders to the microcontroller. This kit includes the microcontroller which is an Arduino 328, as well as a nickel metal hydride

battery and an appropriate charger. The kit also includes an IO expansion board which would allow us to connect more external devices[26].

Whilst it would be useful to have most of the equipment decisions done for us, some of this is not necessary. The IO expansion board for example will most likely be needed, as all we're really interested in adding is a sensor for observations. As well, the price of the kit is quite high given we would also need to purchase a LIDAR sensor on top of it.

3WD 48mm Omni-Directional Triangle Mobile Robot Chassis - £114.34

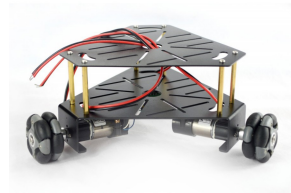


Fig. 8.2: 3WD Omni-Directional Robot Chassis

Unlike the previous chassis this one features only three wheels rather than four. This is still enough to achieve freedom of movement however. Whilst it doesn't include the additional pieces of the kit the other does (such as a microcontroller) we still have our wheels, a supporting structure and appropriate motors with encoders that will provide us with odometric data. There is plenty of room in the middle for items such as our microcontroller and a power source, and the top plate is an ideal mounting point for our sensor[25].

Whilst still a bit expensive, the chassis is still on the cheaper side compared to the previous one and some cost is expected to be incurred given that a 12v DC motor with an encoder can cost around £30. One issue encountered while looking for an appropriate robot chassis is that the majority on sale seem to already include microcontrollers and sensors, which launches their price up and also removes a lot of the choice from the product. Looking around the RobotShop and HobbyTronics website for just an empty chassis only yielded this product. Based on these factors, this chassis will be the one used for the product.

8.1.2 Microcontroller

The main functioning of the robot will need to be controlled by a suitable microcontroller. Something relatively small and cheap with sufficient GPIO pins should work fine.

Arduino Uno Rev3 - £20.80

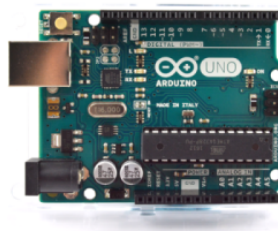


Fig. 8.3: Arduino Uno Rev3

First and foremost the Arduino features a very small form factor, which is good for the robot as less space being taken up by the microcontroller means there is more room for proper cable management and other components. As well, it has several I/O pins which should accommodate any sensor that is used for observation. It also sits at a relatively low price point, costing around £20 on sites such as rs-online[28]. Unfortunately the technical specification is somewhat underwhelming, as it only boasts 30 kB of available memory with an also slow clock speed of 16MHz[29]. The low memory and clock speed could present issues in operation, as it could result in insufficient space to store readings as well as a potentially long time to carry out any read/write tasks.

FRDM-K22F - £23.57



Fig. 8.4: FRDM-K22F

The K22F features higher specs than the Arduino, with 128kB of RAM and a 20MHz CPU. It also features a small form factor as well as an impressive

40 GPIO pins, which will easily accommodate any peripherals that the robot will make use of. The fact that the microcontroller is an arm Mbed product also allows it to make use of the online Mbed compiler as well as the Mbed SDK which could aid in development[1]. Unfortunately the board is not featured on any common vendor websites based in the UK, with most sellers being based in the US which means it could take a while to arrive. Vendors that ship it from the US cost around £23 to £25[?] In addition, it somewhat pales in comparison to a more sophisticated model which can be acquired at no cost from the University's loan office.

FRDM-K64F



Fig. 8.5: FRDM-K64F

Essentially an upgraded version of the FRDM-K22F, the K64F features more RAM and a faster CPU than its more budget oriented counterpart. There are also a few additional peripherals that come already attached, such as a Micro SD-Card reader allowing our robot to read and write to an external storage medium[2]. It too boasts all of the accessibility to the online compiler and Mbed SDK that the K22F does. These features combined with its free availability at the University's loan office make it the ideal choice for the project.

8.1.3 Rangefinder

Based on the discussion regarding the various rangefinding techniques in the mobile robotic's section, it was decided that LIDAR, whilst the most expensive, seems to be the most appropriate for the project. This was primarily due to its ability to be unaffected by poor light levels or other sensors, as well as available sensors that feature pre-implemented functionality. In addition, the high accuracy of LIDAR mentioned by a few papers in 5.3.3 will allow the created map to possess a high level of detail. Here we will look at a few different LIDAR sensors that could be suitable for the project.

RPLIDAR A2M8 360 Degree Laser Scanner - £301.43**Fig. 8.6:** RPLIDAR A2M8

The RPLIDAR A2M8 uses the previous described LIDAR triangulation system, and outputs scan data at 8000 samples per second[30]. It outputs this data via a relevant communication interface, representing scans as distance (what distance between the sensor and the measuring point) and heading (the heading angle of the measurement). It also includes a start flag in its measurement signalling the start of a new scan, which would likely come in useful for processing the scan data as we could simply check this flag to see if incoming data is from a new scan. One of the biggest advantages to employing this sensor would be the SDK it comes with. With full documentation[33], the SDK allows us to access the sensor's entire functionality straight away. It has appropriate methods for beginning the scanning session, ending it and checking sensor health as well as a few other things. This would save a lot of time in the project as we could focus on logical implementation rather than creating the prerequisites that will be needed for the sensor to do what we need it to do.

The major downside to all of these advantages however is the cost. Sitting at just above £300, this is an incredibly steep price for the project if we factor in the cost of the other equipment such as the chassis. This problem leads us to the final choice.

RPLIDAR A1M8 360 Degree Laser Scanner - £93.54

Essentially a less sophisticated version of the previous sensor, this iteration of the M1 LIDAR sensor still offers us a 4000 - 8000Hz sample frequency as well as the previously mentioned SDK[31]. Among a few other things, the key differences here are that we have a slightly slower scan frequency (1 - 10Hz vs the A2's 5 - 15Hz) and a few pieces of SDK functionality that aren't available on the less advanced firmware. The key change is that we lack the express scan function, which is similar to the regular scan but it performs



Fig. 8.7: RPLIDAR A1M8

at the highest sampling rate the sensor can use. The upside however is a lower price point[27].

These downsides are negligible with our product however, ultimately as long as we can easily get observational data at a reasonable rate the project benefits hugely. This functionality at the far more agreeable price point makes this the sensor of choice for the robot.

8.1.4 Software Framework

With regards to the software that will be running on the robot itself, it makes sense to implement it as an operating system. Appropriate tasks will be made to deal with things such as the robot's movement and interaction with the LIDAR sensor. This section aims to explain our potential options for how this operating system will be implemented before coming to a conclusion on which of the options will be used and why.

MBED RTOS

The MBED Real-Time Operating System is an open source operating system for platforms using Arm microcontrollers. It is specifically designed for IoT (Internet of Things) devices, which the OS documentation[3] define as low-powered, constrained devices which require access to the internet.

Aside from the usual features you'd expect from an operating system (threads, semaphores, etc), MBED OS also features C++ based network sockets for the sending and receiving of data, network interface APIs for interfacing with things such as ethernet or wi-fi as well as bluetooth support. One aspect of the Terms of Reference dealt with precisely how map data would be handled, with two of the three options involving wireless transmissions. Should the project go in this direction, MBED OS' numerous APIs for dealing with this would be immensely helpful. All of this may be overkill however, communication without the use of these APIs isn't impossible and if we don't use

wireless communication then all of these features will be of no use anyway. Should another method of dealing with map data such as locally storing it, a much simpler OS would be more appropriate.

μ C/OS-II

μ C/OS-II (which will hereon be referred to as Micro C OS) is a very simple real time operating system designed for use with embedded systems. Labrosse[17] discusses a number of different features of Micro C OS in the documentation in addition to ones you would typically find in an operating system (semaphores, task management, etc). Labrosse discusses how Micro C OS is able to be used on a wide variety of microprocessors owing to its design, allowing it to be highly portable. In addition, Micro C OS's scalability is discussed, allowing for only the services required in the operating system's host application to be used. This feature in particular would come in useful for the project given the use of a microcontroller, as it would allow us to cut down memory usage to only what we need which should help performance issues from arising.

8.2 Techniques

8.2.1 Acquiring Scan Data

One aspect that needed to be addressed was how the observational data was actually dealt with. Once the LIDAR is taking observations, how does this data find its way back to a terminal where it can actually be used? In the Terms of Reference, a few different options were considered. These were either storing the data locally on the drone or wirelessly transmitting the map data. What follows is a brief investigation into which of these two approaches would be the most viable.

Local Storage

One approach is for the observational data taken by the sensor to be stored on a local storage medium. The K64F features a Micro-SD card socket, and given that the Micro-SD card can be obtained from the University's loans office this approach would incur no extra cost or time spent waiting for a delivery. Mbed features an SDFileSystem library[23] which would allow for a quick implementation of this on the software side of things. The use of local storage would also allow for the robot's range to be essentially unlimited as well.

The speed of this approach should be considered. Given how it's likely the robot will take at least a few thousand readings based on the chosen sensor's sample rate, writing thousands of values to the SD Card might end up taking a while. Given the instant accessibility of this equipment, a few tests were carried out to see if these issues might present a problem.

A simple program was used that measures how much time it takes for a section of code to execute. A 60MHz timer was started and stopped before and after the execution of code that involved writing 16000 samples to a file on the Micro SD-Card, which given the LIDAR's sampling rate of 4000 - 8000 was a few second's worth of scan data. An actual time in microseconds for the software's execution can be determined by dividing the elapsed clock cycles by the timer's clockrate in MHz. This code was ran multiple times and the mean clock cycles for each execution time was stored. Table 8.1 shows the results of these tests.

Test	Time in Seconds to Write Data
1	7.9
2	7.9
3	7.7
4	7.7

Tab. 8.1: Time taken to write 16000 dummy samples to the Micro SD-Card

Writing several thousand samples to the SD-Card won't be an issue, and the created files seemed to be around 250kB meaning the 8GB SD-Cards that can be freely obtained from the University's loans office will have more than ample storage capacity.

Wireless Transmission

In order to transmit the map data, some form of transmitter will likely need to be added to the microcontroller. An RF (Radio Frequency) transceiver could be added to the microcontroller allowing it to transmit and receive radio signals. The K64F features headers for use with 2.4GHz radio add on modules, and a nRF24L01P Nordic transceiver can be obtained very cheaply for just a few pounds online. The Mbed website also has a library[16] for this transceiver which would allow for interaction with it to be relatively straight forward. Data transmission shouldn't be a problem either. The transceiver's data sheet?? says that the transceiver has a minimum transfer rate of 250kbps. The A1M8's documentation[31] mentions distance and heading as two attributes of the working mode's data output. It also mentions the sensor's 4000 - 8000hz sampling rate. If we assume the sensor is always going to be working at the highest sampling rate and that both the

angle and distance values are stored as floats, then every second we receive 16000 float values (8000, the sampling rate, multiplied by two since each scan will feature two floats to be stored). Since each float is four bytes that means to acquire this second's worth of scan data, the transceiver would need to transmit 64000 bytes, or 64 kilobytes. Given the worst case scenario transmit speed of 250kb/s this would be done in less than a second, so we shouldn't have any trouble transmitting as much observational data as we might need.

The problem here though is that having just a transceiver on the robot's microcontroller wouldn't be enough for interaction with it, a second microcontroller would need to also have a radio chip soldered on so that something could actually receive transmissions. Attempting to get pairs of microcontroller listening to each other presents problems in of itself, and as well the use of a radio would mean the robot's range is potentially limited.

Deciding on an Approach

Whilst the wireless transmission approach boasts more initial range and allows for the robot's observational data to be acquired much quicker, the readily available components, reduced setup time and greater internal storage offered by the Micro SD-Cards space make it the technique of choice for dealing with the observational data. Despite this, the ability for a transmitter to send a simple confirmation signal indicating the robot isn't stuck or experiencing problems would greatly augment the robot's ranged capabilities. Should time allow it, this could be a good addition to the robot once the rest of the system is in place.

8.2.2 Dead Reckoning

Odometry is the usage of data from motion sensors to estimate an object's position. One such implementation of this that will be looked at for the project is dead reckoning. Dead reckoning tracks the robot's position by using data from wheel encoders that count the number of wheel rotations performed during operation. From this, the internal tracked position of a robot can estimate its new position after periods of movement. Given that the chassis being used for the project has wheel encoders as part of the wheel motors, this approach would allow us to perform odometry without needing the use of additional kit. Dead reckoning is not without issues however. Most pressing it does not account for wheel slippage. If the robot has a poor grip on the ground and the wheels slip, then the wheel rotation doesn't accurately correspond to the robot's location [9]. These issues would compound as well. As more and more slippage happens, the

robot's internal position would become less and less accurate to where it actually is.

9. CONCLUSIONS

The literature review has allowed the scene to be set for the project now. The research and evaluation of relevant literature on mobile robotics will be incredibly valuable moving forward in the project, and has allowed for an understanding of the field that should aid development. In addition, the SLAM investigation has made clear the nature of the issue that will be dealt with when it comes to attempting to get the mobile robot to map the environment as well as providing a potential solution to the problem.

The review of tools and techniques will be invaluable to development as well. Moving forward, it will be of great benefit to truly understand the tools that are being employed during the project as well as some of the different techniques that can be used to solve issues that come up.

Part II

SYNTHESIS

10. INTRODUCTION

Now that the relevant product fields are understood to a sufficient degree and the product's needs have been established we can begin to plan out how we will achieve the project's objectives. First we will establish the methodological approach that will structure the product's development. Then, for each of the core product requirements we will look at a high level design overview of how the requirement will be met, the actual implementation of this design, and testing to evaluate how well this requirement has been satisfied by this implementation.

11. METHODOLOGICAL APPROACH

The project employed a prototyping methodology for development. First, we must perform an investigation into the system requirements, which was performed in the analysis with a list of specific core requirements being outlined in the conclusion. Following this the first prototype is built, which will resemble an incredibly basic scaled down version of how the final system should ideally look. This prototype is then thoroughly evaluated, with potential changes that would bring the prototype closer to the final system being figured out. Finally, these changes are then implemented and the prototype is evaluated once again. This is repeated until the product has reached its ultimate goals.

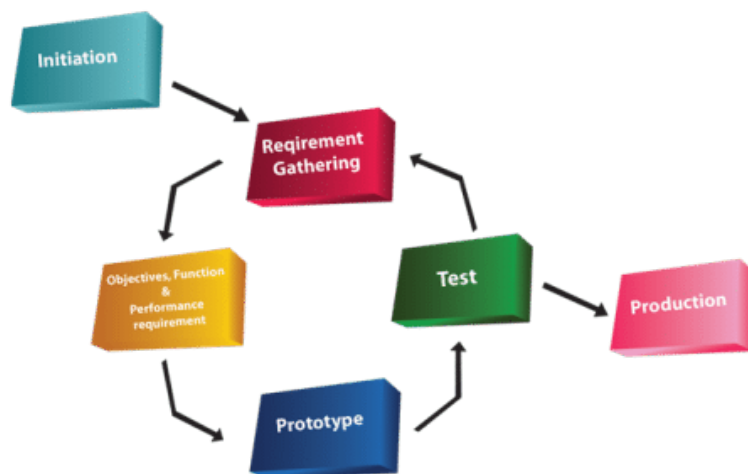


Fig. 11.1: Prototyping Methodology

This methodology was employed partially due to the hardware nature of the product. Key features of the robot hinged on aspects like movement and observation being possible, so it was important to get a prototype that has this basic functionality up and working and quickly as possible. The prototyping approach allowed a pure focus on getting these functions working,

and once they were the robot could be improved iteratively.

11.0.1 Initial Prototype

As previously stated, this methodology involves an initial prototype that all future prototypes will be an iteration of. The initial prototype of the project will involve the basic, primary construction of the robot. This will first involve the basic chassis assembly followed by attempting to connect relevant components to each other. Once this is done, we'll hopefully have an incredibly crude robot that won't be capable of anything, but should hopefully serve as a solid foundation for future development toward the project goals.

A similar approach will be followed for processing the map data. It won't be developed simultaneously at first as the robot needs to be capable of storing observational data from the LIDAR before the program can actually do anything, however it will start to be put together once the robot crosses this developmental threshold.

12. DESIGN

12.1 Introduction

Before implementation of the specific project aims can be implemented, we first need the initial prototype that we can begin iteratively improving. Therefore, it's logical to design the initial robotic prototype first. What follows is an overview of how the initial robot prototype will be designed, followed by a breakdown of each of the individual project aims will be met.

12.2 Initial Overview

Here will be the design for a high level overview of the project, first establishing the basics of how the robot's structure will be assembled and how the robot's core software will be made.

12.2.1 Hardware

Chassis

First off will be the basic assembly of the three wheeled omnidirectional chassis. The chassis is comprised of two triangular metal plates, which will be joined together by screwing metal rods into pre drilled holes on each of the platforms. The lower platform is where the motors and mounting points for the omnidirectional wheels are found, so once the plates have been connected the wheels will be pushed onto these mounting points and locked into place. Fig 12.1 shows the assembly document that came with the chassis.

Now it's time for the microcontroller setup. The microcontroller used in this project is the FRDM-K64F, chosen for its free availability from the University loan office, small form factor and compatibility with the Micro C Operating System which is being employed for the robot's core program. To control the robot we'll also need a motor driver. The dfRobot Quad Motor Driver Shield is being employed for this project, it can be obtained

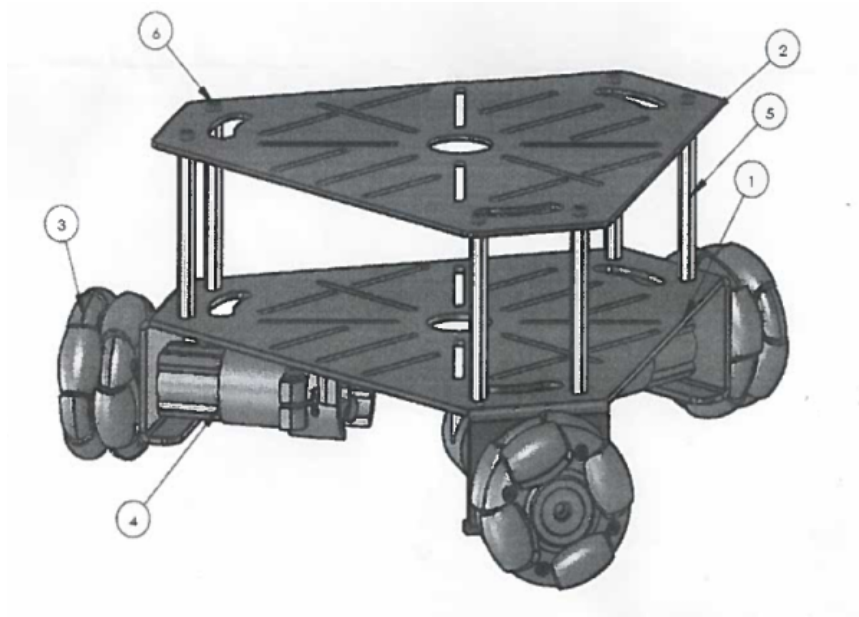


Fig. 12.1: Chassis assembly guide

from Amazon for around £13. It was chosen for this relatively low cost, the easily available documentation and the fact that it can easily plug into the FRDM-K64F, the microcontroller of choice for this project. This shield will be lined up to the appropriate pins on the K64F and plugged in, in preparation for using the microcontroller to drive the motors.

The assembly shouldn't be overly complex, no soldering should be needed to connect components so it should be mostly a case of fitting things together.

Power

The hardware components that will need powering are the three motors used to drive the omnidirectional wheels, the microcontroller, and the LI-DAR sensor. Table 12.1 has a run down of these hardware components, the voltages, and the milliamp consumption that they have.

Component	Voltage	Milliamp
3x DC Coreless Motor	12V	Up to 1400 mA
FRDM-K64F	5 to 9V	50 mA
RPLIDAR	5 to 10V	Up to 1050 mA

Tab. 12.1: Components and respective voltages

To save on cost and complexity, it would be ideal to only use a single battery

for the robot's power source. A single power source will be used for the robot, a battery holder containing 8 1.5V batteries will give us the 12V we need for the motors. Then, a step down voltage regular will be used to lower the voltage required for the other components. A 7v and two 5v currents will be needed to supply power to the microcontroller and the LIDAR respectively.

Generally cheaper alkaline batteries have about 1800 to 2800 mAh (milliamp hours) in them, so an 8 pack of these will give us about half an hour's worth of operation before the robot starts to suffer due to low power. This is a worst case scenario assuming a constant high power draw, but this should be sufficient for the purposes of prototyping and demonstration.

12.2.2 Software

Robot Program

The robot's software will be a Micro C OS, implemented using C++ classes and compiled with the mbed SDK platform. Classes will be created based on grouped functionality to allow for greater code readability and efficiency. Once development begins, it would be prudent to create a skeleton OS based on the classes and states outlined by the documentation in 12.4. This will enable confirmation that the software is able to be deployed on the microcontroller, and will serve as a good base for functional implementation.

12.3 Designing for Requirements

Now that the fundamentals of the hardware's construction and the software's foundations are understood, we can begin to move toward truly fulfilling the project requirements. What follows is an overview of how each of the different project requirements will be met, with appropriate explanations toward the hardware and software employed.

12.3.1 Movement

The first of the three functional requirements outlined in 7.1 is that 'the robot must be capable of movement'.

Hardware

The dfRobot Quad Motor shield is being used to help control the motors. Fig 12.2 shows an overview of the shield.

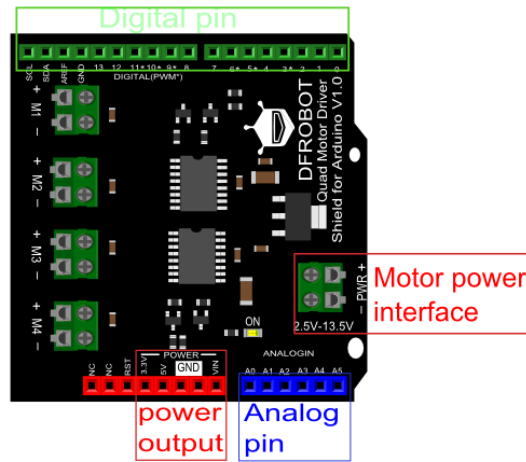


Fig. 12.2: Example shield setup

The motor power interface is how the shield will actually receive its power from the battery. The power and ground cables from each of the chassis motors will be plugged into the appropriate motor pins that can be seen on the left of the shield figure. Fig 12.3 from the shield's documentation shows an example of this setup with four motors.

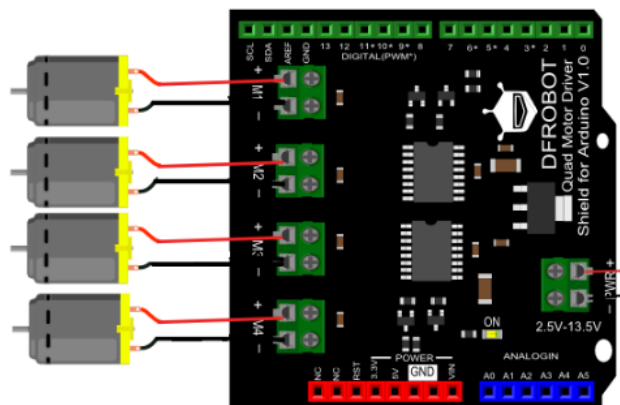


Fig. 12.3: Shield connection

Each of the motors power and ground cables will connect to the appropriate pins on the shield. The motors can be manipulated once they receive power in this fashion. The shield is able to affect the electricity being sent to the

motors by changing its polarity and by using pulse width modularity. By changing the pin to HIGH or LOW, we can affect the direction that the motor spins in (forward or backwards) and pulse width modulation allows us to affect the speed of the motor.

Software

For each motor on the shield, there are two associated pins. The first of these pins controls the motors direction (forward or backward). The second of the pins controls the motor speed, achieved using pulse width modulation. To manipulate the motors these pins will need to be declared as variable within the robot's software and assigned different values based on what we are trying to do. The directional pins are either HIGH or LOW, so they will be declared as digital pins that are assigned a binary value. For the speed pins, mbed features a PwmOut interface with a .write() method that can be used to assign a duty cycle value to the variable. This is perfect for what we need. To actually steer the robot, methods in the control class will receive angle and speed values and perform trigonometrical calculations to find out what speed and direction values need to be sent to each of the three motors.

12.3.2 Observation

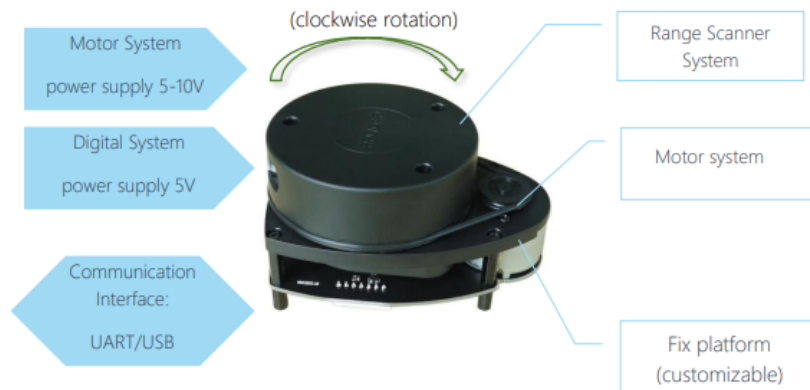
The second of the three functional requirements outlined in 7.1 is that 'the robot must be capable of observation'.

Hardware

Only one range finder is being used to gather the observation data, the RPLIDAR A1M8 LIDAR sensor. The sensor is composed of a platform with a motor system that spins the range scanner as it takes readings, as well as some pins that can be used for communication. Fig 12.4 illustrates these components.

There are seven pins on the underside of the LIDAR sensor, shown in fig 12.5. These pins need to be connected to the appropriate microcontroller ports if the LIDAR is to work.

The GND pins are simple ground pins, they will need to be connected to ground pins on the microcontroller. The RX and TX pins (Recieve and Transmit respectively) are serial pins that will be used for communication with the microcontroller. These LIDAR pins will be linked to their opposite counterparts on the microcontroller (LIDAR RX to Microcontroller TX and vice versa). The V5.0 and VMOTO are simple power pins, they will powered

**Fig. 12.4:** LIDAR Underside**Fig. 12.5:** LIDAR Underside

via a stepped down 5V current taken from the voltage regulators. This is how the LIDAR sensor will retrieve power. The MOTOCTL pin is the motor control pin that listens for a signal indicating that a connected device is ready to receive data. The signal is either high (ready) or low (not ready). To make use of this pin, a generic GPIO pin from the microcontroller will be configured within the microcontroller's software and set to 1 (high) when the robot needs to begin taking scan data.

Software

SLAMTEC have a document detailing the LIDAR protocol[32]. This details the specifics behind how to communicate with the LIDAR. The protocol works by receiving request packets made up of certain bytes. Depending on what the bytes are, the LIDAR interprets the packet as a different com-

mand. All request packets contain a one byte start flag and a one byte command field, with optional fields for things like a payload depending on the request being sent. Depending on the command, the sensor will respond in a different way. Fig 12.6 shows the documentation’s outline of a simple SCAN request which causes the LIDAR to begin outputting observational data.

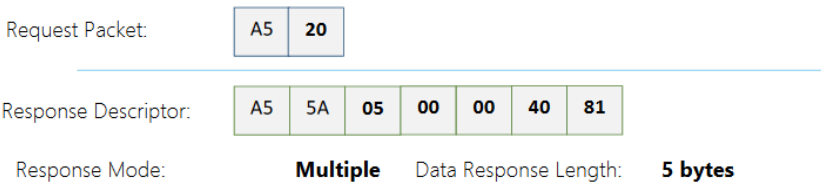


Fig. 12.6: Scan Request

To begin scanning, a request packet with two bytes of the listed values would need to be sent to the LIDAR. It’s advised by the documentation to wait at least 2ms after this before sending any further requests to give the sensor a chance to process the request. Once the sensor has processed the request a response descriptor is sent back. All response descriptors begin with two start flags, and then have five bytes providing information about the coming responses. It provides the size of a single incoming data packet, the send mode of the current request/response session (e.g. single response to the request or multiple responses) and the data type of the incoming response packets.

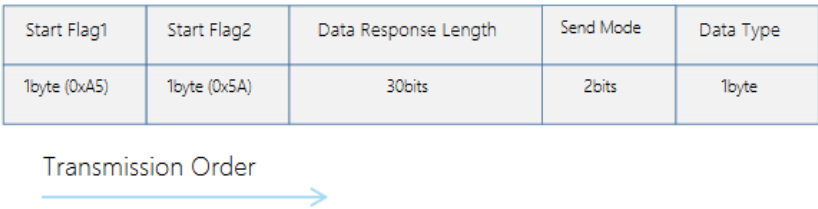


Fig. 12.7: Response Descriptor

This could be implemented through the use of data structres in the robot’s program. For example, a scan request data structure with two fields (start flag and command) could be initialized and then sent to the LIDAR sensor via the serial connection. Data structures resembling the response descriptors could be populated with information sent back from the sensor as well, allowing for easy processing and manipulation of retrieved data.

Once data is received from the LIDAR, the protocol documentation states how it can be understood. In each byte returned by the LIDAR, the protocol

states a binary breakdown of what each chunk of bits represents. An example of this can be seen in fig 12.8, which is an illustration of the response to a scan request. So for example here, the first 6 bits of the first byte pertains

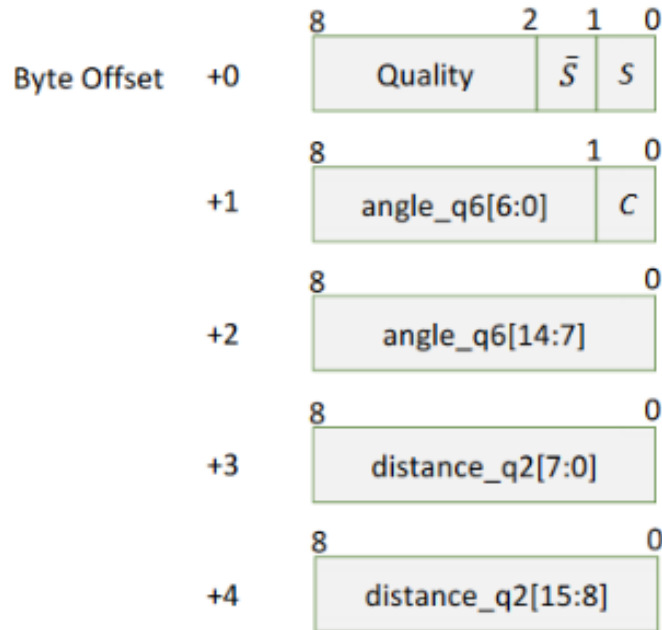


Fig. 12.8: Scan Response Breakdown

to the readings quality attribute. The 16 bits from the final two bytes are the first and second halves of an overall value which is the distance value, etc. The bytes will need to be changed into their binary representation for this to separate out individual bits that mean different things.

This may not be needed however. SLAMTEC provide an SDK for the LIDAR sensor which comes in the form of header files which will automatically implement this functionality. Table 12.2 has a manifest of the SDK and the functionality that it provides.

File	Purpose
rplidar.h	Parent file for subsequent header files
rplidar_driver.h	Provides RPLidarDriver class for interfacing with sensor
rplidar_protocol.h	Defines structs and constants for the LIDAR protocol
rplidar_cmd.h	Defines request/answer structs for LIDAR protocol
rptypes.h	Platform independent structs and constants

Tab. 12.2: RPLIDAR SDK files

All that would be needed is for the C++ program running on the microcon-

troller to create an `RPLidarDriver` variable which would be used to represent the connected LIDAR sensor. Once this is achieved the premade methods that implement the protocol functionality could be ran to achieve control over the LIDAR. Once the LIDAR is retrieving data the class will be able to populate a buffer with readings which can be written to a file once it's full. Readings could also be checked by the robot's main program to see if a distance measurement is particularly low, and if it is the robot could steer away from whatever angle the distance has been measured from to avoid obstacles.

12.3.3 Processing Observational Data

The last of the three functional requirements outlined in 7.1 is that 'the observational data must be processed into a map'.

Hardware

Once the robot is receiving data from the LIDAR sensor, it needs to be transferred to an external program where the CSM software can be ran to generate a map from it. Based on the results of the previously conducted investigation into what would be the most suitable way to store observational data, a Micro SD-Card will be used. The FRDM-K64F has a Micro SD-Card socket attached to it, and the microcontroller will save the observational data to this card so it can be plugged into a machine where it is able to be processed. The University's loans office can provide an 8GB card as well as an adapter allowing it to be plugged into a PC's USB slot, so this approach won't incur any extra cost. The saved data will only be angle and distance measurement pairs, so the card's (relatively) small size won't pose any memory issues.

Software

First, the microcontroller must save this data to the connected Micro SD-Card. Mbed has a library called `SDFileSystem` used for interaction with connected SD cards. Using this library, a simple text file will be created on the Micro SD-Card and angle/distance measurements obtained from the LIDAR sensor will be written to it. As the LIDAR scans data it will populate a buffer, and once the buffer has been filled it will be written to the card. After this the buffer will be cleared and begin being populated again. The FRDM-K64F features 256kB of RAM, and a C float is around 4 bytes. If we quite generously assume 156kB will be taken up by task stacks and other program necessities, that still leaves around 100kB of free RAM. In 100000

bytes, we can still store 25000 floats in RAM. Given that each returned scan will probably have two floats (an angle and a distance) we can store at least 12500 readings before the robot needs to pause to write them to file, which should be more than adequate.

Once readings are successfully being obtained and stored to the SD-Card medium, development should focus on turning this data into a map. The first stage will consist of something relatively simple. Converting the measurements into a simple outline or scatterplot graph will allow for some easy visibility on the quality of the readings being retrieved, and will serve as a solid base prototype as developmental focus moves toward making this generated map incorporate multiple scans at different positions.

To design the GUI, a Python library called Tkinter will be used. It offers all the widgets that will need to be used to make a basic GUI such as frames, buttons and labels and its support of most (if not all) operating systems will allow it to be easily used. To create the regular map to begin with, a Python library called matplotlib will be used. Provided with x and y coordinates, matplotlib can plot these points onto a scatterplot graph. If the angles and distances are converted into appropriate x and y coordinates it should be relatively straight forward to produce a map early on. Once this has been achieved, focus should be turned to implementing CSM for a true SLAM product. The GUI's job at that point is to create an appropriate input for CSM and to use it to generate a map from the readings. CSM takes in readings in the form of either a json or a carmen log. Of the two, json will be used due to its familiarity and much more easily accessible documentation around the internet. To invoke the CSM software itself via the program, Python features a subprocess module which allows for Python programs to spawn simple subprocesses and retrieve information they return. Once the software has processed a map from the readings, it will be output onto the screen so that the user can view it.

12.4 Final Design Documentation

12.4.1 Class Diagram

Fig 12.9 shows the intended classes for the robot's software.

- **Robot** - The core of the robot's program, this will contain the initial OS setup and the main task which will delegate commands to the Control and Lidar classes.
- **Control** - The Control class will deal with manipulation of the robot's motors. It will contain the relevant variables for each of the motor's direction and speed pins, and given an angle and speed will calculate the necessary values that need to be assigned to these pins to steer the robot in the right direction.
- **Lidar** - The Lidar class will handle the robot's operations with the sensor. It will store relevant variables for the sensor's operation such as the DTR pin variable allowing it to signal that it is ready to receive data from the sensor, and relevant serial variables to actually retrieve this data from. It will also contain a buffer which it will populate with readings before having them written by its fileWriter class. It will have methods to start and stop the LIDAR's scanning, a method to actually store the readings in the buffer and a method that returns the latest reading for obstacle detection. It will also have some methods that act as wrappers for the SDK's functionality such as `getHealth()` and `getInfo()`.
- **FileWriter** - The FileWriter class deals with writing the data to the robot's Micro SD-Card. It will have a declaration of the `SDFFileSystem` class provided by the library of the same name, and when supplied with a populated `readingsBuffer` by the Lidar class it will write the values in the buffer to the SD-Card.

12.5 State Machine

The finite state machine in 12.10 represents the different states that the robot can be in.

- **Initialize** - Setup of variables and tasks once the robot receives power.
- **Scan** - Retrieving observations from the LIDAR and populating buffer.

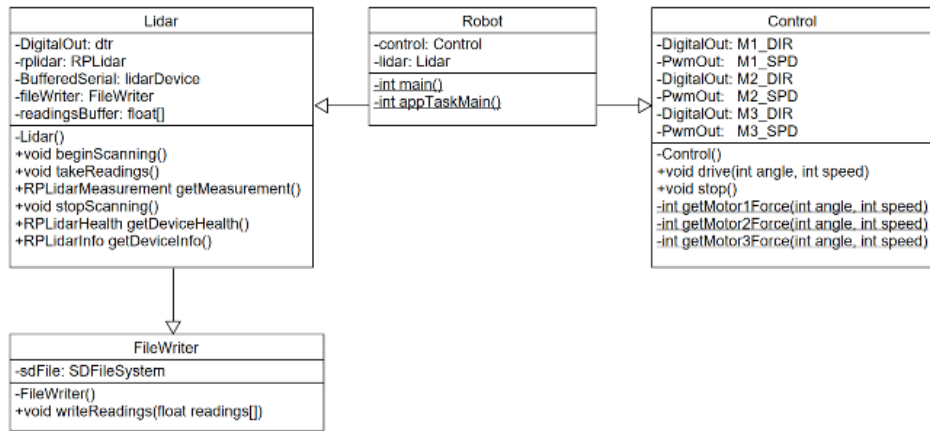


Fig. 12.9: Class Diagram

- **Obstacle Detected** - Received measurement has a very low distance.
- **Save Data** - Once the buffer is full it's written to a file.
- **Move** - Moving away from obstacles.

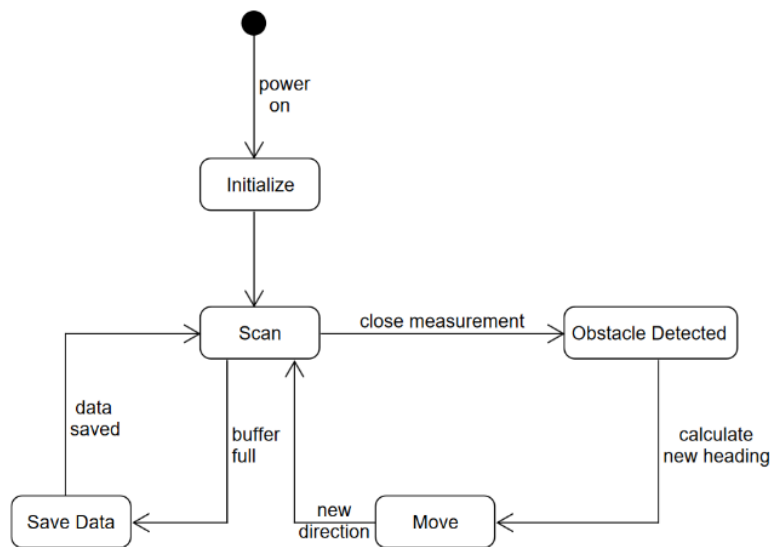


Fig. 12.10: Finite State Machine

13. IMPLEMENTATION

13.1 Introduction

This chapter will describe the actual creation of the product. The implementation of relevant software and hardware features for the different product components will be discussed, as well as unforeseen problems that were encountered.

13.2 Initial Setup

Before work can begin implementing any specific functionality for the robot, some preliminary hardware assembly and software setup had to be carried out.

13.2.1 Hardware

First of all the physical chassis had to be assembled. The available documentation for the chassis' construction is 12.1 which came in the chassis box. Construction was relatively straight forward and the diagram was of help with a few parts, so it only took around thirty minutes to assemble the structure. Once it was built, the wheels where pushed onto the motors where they clicked on. The Quad Motor Shield was then installed onto the K64F microcontroller without any problems.

13.2.2 Software

A basic Micro C OS was created and deployed onto the microcontroller. This OS resembled the pseudo code and was composed of skeleton tasks and classes with method stubs.

13.3 Implementing Requirements

13.3.1 Movement

Hardware

First the dfRobot Quad Motor shield was fitted onto the K64F microcontroller and placed in the lower level of the robot. Each of the three wheel motors had a ground and a power cable attached to them, these were all fed through the hole in the centre of the lower plate and connected to the appropriate ports on the motor shield. In the interest of keeping the components tidy, the power and ground cables were fastened to the lower plate with zip ties.

Each pair of power and ground ports are labelled with a number on the shield (e.g M1, M2 etc). From hereon in, the motors will be referred to with these same numbers. Fig ?? shows the position and names of each of the wheels on the robot.

At this point it was time to power the motors by connecting the 12V battery pack to the motor shield. Initially it was planned that this would be achieved from a breadboard, as the breadboard would allow the use of voltage regulators to supply lower voltages to the other components. A problem was encountered here though in the way of space. The microcontroller, motor power cables and battery pack alone already cluttered nearly all of the lower level of the robot. The additional of a breadboard with voltage regulators and additional wiring would be an incredibly difficult fit with wires potentially breaking or coming loose, and there was still the matter later on of connecting the LIDAR properly. This resulted in two decisions being made. The first was that the LIDAR would simply receive power from the microcontroller. The second decision was that the microcontroller would receive its own battery. The K64F features a VIN pin for power that takes between 5V and 9V, so a 6V battery pack was obtained. The power cable was connected to the VIN, and the ground cable shared a ground port with the motor's battery pack. Preliminary tests confirmed that the motors responded to commands sent to them by the robot's software.

Software

Each of the three motors were now connected to two pins on the Quad Motor Shield. One pin affected direction, the other affected the speed. Direction was a simple case of polarity. Depending on the motor, HIGH or LOW dictated whether the motor spun forwards or backwards. These pins could be declared as DigitalOut variables, and assigned either a 1 or a

0 depending on what was needed. The pins that affected speed did so via pulse width modulation, so these could be declared as PwmOut variables and floats could be assigned to them to affect their duty cycle. The motor driver's documentation has a table stating which of the pins on the shield corresponded to which motors, so whichever K64F pin was connected to the relevant shield pin was used for the variables. These variables were declared within the control header and cpp files.

```

DigitalOut  M1_DIR(D4);
PwmOut      M1_SPD(D3);

DigitalOut  M2_DIR(D12);
PwmOut      M2_SPD(D11);

DigitalOut  M3_DIR(D8);
PwmOut      M3_SPD(D5);

```

Stubs of the methods listed in 12.9 were created for the class. Some basic movement methods were made to ensure the robot could actually navigate, making it go forwards and backwards but the ideal implementation involved movement being possible for every heading in a 360 degree radius. Some preliminary problems with calculating force values came up as an issue however. Most force calculations resulted in a raw float values between -1 and 1, but this couldn't be assigned to a speed spin because the pins only supported positive value. Forward and backward motion was dictated by the directional pin. Issues involving other project components ate up time as well, and this aspect of the project was accordingly never fully realised. These problems are explained in 17.

13.3.2 Observation

Implementation of the robot's ability to take observations about its environment first involved a hardware configuration. The LIDAR sensor needed to be connected to the microcontroller so that it could draw power from the battery, as well as communicate with the microcontroller allowing it to receive commands and send observational data. Once the hardware connection was established, software on the microcontroller had to be capable of operating the LIDAR sensor and receiving its observations.

Hardware

As previously discussed, the seven pins on the underside of the LIDAR sensor need to be correctly connected up to the appropriate microcontroller pins if the sensor is to be used.

Table ?? gives an overview of which LIDAR pins needed to be connected to which microcontroller pins.

LIDAR Pin	K64F Pin	Pin Purpose
GND	GND	Ground
RX	TX	Serial Communication
TX	RX	Serial Communication
V5.0	5v	LIDAR Core Power
GND	GND	Ground
MOTOCTL	D7	DTR (Data Terminal Ready)
VMOTO	3v3	LIDAR Motor Power

Tab. 13.1: LIDAR Microcontroller pin setup

Due to the dfRobot Quad Motor shield being plugged into the K64F microcontroller, these LIDAR pins need to be connected to ports on the robot shield that are plugged into the appropriate microcontroller pins. Essentially, we look at what port needs to be used on the underlying microcontroller, then plug it into the motor shield port that is sitting on top of it. The LIDAR pins and shield ports were connected using simple male to female connector wires.

With regards to power, it was previously mentioned in the movement implementation that due to spacial constraints the LIDAR would receive power from the microcontroller. The microcontroller features two pins that can supply a voltage, a 5V pin and a 3V3 pin. Both pins would ideally receive 5V, but to save room and ensure a simpler robot the scanner motor was connected to the lower voltage pin. It was decided that the scanner motor should receive the lower voltage because it only appeared to make the scanner move a bit slower, whereas there was a concern that supplying the actual LIDAR core with a lower voltage might result in problems performing the actual scanning.

Once this was done, the LIDAR was able to function mechanically.

Software

Following the hardware setup the microcontroller needs to actually communicate with the LIDAR sensor so that it can send commands, as well as receive observational data. This was the responsibility of the lidar.cpp class.

To first of all ensure the LIDAR is ready to communicate, the previously discussed MOTOCTL pin on the LIDAR needs to be receiving a HIGH signal so that it knows the microcontroller is ready to begin receiving data. The K64F features numerous GPIO pins that can be easily configured for

usage in situations like this. One such pin (D7) was declared as a basic `DigitalOut` allowing the microcontroller to manipulate the pin's polarity.

```
DigitalOut dtr(D7);
```

Simple methods were used for this manipulation. When the sensor needs to output data the pin was set to HIGH...

```
void beginScanning() {  
    dtr = 1;  
}
```

...or when it had to stop, it was set to LOW.

```
void stopScanning() {  
    dtr = 0;  
}
```

The microcontroller didn't instantly begin receiving data once this was set up however, as simply connecting the LIDAR and microcontroller serial pins are not enough for communication. In order to interact with the serial channel it needs to be declared in the robot's core program as an appropriate variable. There exists a library for arm MBED microcontroller called `BufferedSerial` which can be used for this. The original `Serial` library that it is based on allows for serial communication, with `BufferedSerial` simply adding a buffer to this. The core functionality remains the same however.

First the pins being used for serial communication need to be declared as a `BufferedSerial` variable, with `BufferedSerial` taking the TX and RX pin (in that order) of the microcontroller. It also has an optional baud rate parameter, the LIDAR documentation lists the used baud rate as 115200 so this was supplied just to be safe.

```
BufferedSerial lidar_device(D1, D0, 115200);
```

As previously discussed in 12.3.2, SLAMTEC has an SDK that automates most of the LIDAR functionality. The appropriate files were added to the project and made able to be used by the core program with a simple include statement at the top of the `lidar.cpp` file. Following this a base `RPLidar` variable can be declared from a class the library adds.

```
RPLidar lidar;
```

Once this variable has been declared, in the class' main method the DTR is set to 0 to ensure that the LIDAR won't start scanning and outputting data when we don't need it to. The LIDAR variable is then connected to the relevant serial variable (the previously declared `BufferedSerial`) and the start scan command is issued. Despite receiving the command to start

scanning the LIDAR won't begin until the DTR gives it the go ahead, but by configuring this as soon as the program begins it's a lot easier to stop and start the LIDAR data output.

```
Lidar::Lidar() {
    dtr = 0;
    rplidar.begin(lidar_device);
    rplidar.startScan();
}
```

Now that the LIDAR is configured and can begin sending out data, it's time to start actually storing it within the program. First we need appropriate mediums for the storage of LIDAR data. This entails something to store a LIDAR sample that has just arrived so we can manipulate it to take what we need from it, and also a buffer to store all the total readings collect so that they can later be written to disk.

In order to store a newly receiving reading, the RPLidar SDK implements a basic struct that can be filled in with a `.getCurrentPoint()` method. This will be used so that received readings can be easily manipulated.

```
struct RPLidarMeasurement
{
    float distance;
    float angle;
    uint8_t quality;
    bool startBit;
};
```

Now a buffer is needed to store the readings so that they can be written to disk later on. A large number would be preferable so the robot doesn't need frequent stops to save them to the SD-Card. Initially a two dimensional array was used but there was some difficulty in supplying it as a parameter to the `fileWriter` class, so two large arrays of floats were used.

```
float angles[8000];
float distances[8000];
```

Now that readings can be manipulated and stored, it's time to properly implement the method that stores them to a buffer.

```
void takeReadings() {
    struct RPLidarMeasurement measurement;
    for(int i = 0; i < sizeof(angles); i++) {
        lidar.waitPoint();
        measurement = lidar.getCurrentPoint();
        // Get angle
```

```
        angles[i] = measurement.angle;
        // Get distance
        distances[i] = measurement.distance;
    }
}
```

The method iterates through both buffers and stores the angle and distance of each reading it receives.

13.3.3 Processing Observational Data

Processing the observational data first involves actually finding a way to take it from the robot. It was determined that the best way to implement this would be via the use of a Micro SD-Card. Once this was done, the data itself needed to be turned into some form of map.

Hardware

The only hardware elements that were needed to implement this functionality were the Micro SD-Card socket on the microcontroller, a Micro-SD card and a USB adapter allowing for a computer to access the files on the SD-Card. The socket was already attached to the microcontroller and no adjustments needed to be made to the hardware for the microcontroller to access it.

Software

A FileWriter class was created to handle the functionality of writing the data in the buffer to disk. In order to access the Micro SD-Card the SD-FileSystem library was used, and was incorporated into the program by simply including the main SDFFileSystem header. To use the library, an SDFFileSystem variable first needs to be instantiated. This variable takes five parameters, four relevant microcontroller pins and a name. Table 13.2 describes each of the four pins the SDFFileSystem needs, the name of the corresponding microcontroller pin and what the pin actually is.

The code risks being unclear if regular pin names are always used, so the relevant pins were first defined into more understandable names. The SD-FileSystem variable was then declared.

```
#define MOSI    PTE3
#define MISO    PTE1
#define SCLK    PTE2
```

SDFFileSystem Parameter	K64F Pin	Pin Purpose
MOSI	PTE3	Master Out Slave In
MISO	PTE1	Master In Slave Out
SCLK	PTE2	Serial Clock
CS	PTE4	Chip Select

Tab. 13.2: SDFFileSystem Setup

```
#define CS    PTE4
```

```
SDFFileSystem sd(MOSI, MISO, SCLK, CS, "sd");
```

With this variable the file system on the Micro SD-Card is able to be accessed. The previously described scan task is what populates the readings-Buffer with data that is needed, so a method is required to iterate through the buffer and write the observational data to the Micro SD-Card.

```
void writeReadings(float angles[], float
    distances[]) {
    // Create the readings file on the Micro SD-
    Card
    FILE *fp = fopen("/sd/readings.txt", "w");

    // Error checking
    if (fp == NULL) {
        pc.printf("Unable to access/create file \n");
    }

    // Iterate through the readingsBuffer and write
    the angle/distance pairs
    for(int i = 0; i < angles.length; i++) {
        fprintf(fp, "%f %f", angles[i], distances[i]);
    }

    // Close file
    fclose(fp);
}
```

First the fopen method is called with a designated file, and the "w" parameter (meaning write) ensures that the file will be created if it doesn't already exist. Then a basic error check follows to make sure that the file is accessible, followed by an iteration through the readingsBuffer. As was previously mentioned, the buffer is populated by pairs of angle and distance measurements, so for each entry in the array the angle and distance is written to

the file. These pairs were written as two values separated by a comma, with each value being separated by a space. This was done to make it easier to read from a program point of view. The plan was that the file can be split by spaces to get pairs, and then each pair can be split by a comma to get the individual values. Finally, the file is closed.

As mentioned in 12.3.3, a GUI would be used for the processing of map data. Tkinter is a simple Python package that is capable of quickly creating incredibly basic GUIs. All that was needed was a panel with some buttons that called methods to turn our readings file into a map.

The first thing to do was begin generating a plain map. Once a simple map could be made from angle and distance measurements obtained from the robot, this process could be elaborated upon to move toward something more thorough generated by CSM. For this, matplotlib was used. matplotlib would allow the generation of a scattermap and, when supplied with x and y values, would plot appropriate points at these coordinates onto the map. The first task was to convert angles and distances into usable x and y coordinates.

Initially dummy data was used to verify that the process worked. First, data from a .txt file had to actually be read into the program. A button was attached to a simple method that did this.

```
# Retrieve distance-angle pairs from a text
file
def readData(self):
    try:
        with open("coordinates.txt") as textFile:
            lines = [line.split() for line in textFile]
            # Values are read in as strings, so we
            # convert them to ints
            for i in lines:
                i[0] = int(i[0])
                i[1] = int(i[1])
            return lines

    except IOError:
        print("Error reading file!")
```

Angle and distance pair values are read in from a text file. These values are separated by whitespace, so the method iterates through each line in the file and splits each line into two tokens which are kept in an array. These tokens are read in as strings so they are cast to ints.

Once this data was being gathered from a text file it needed to be turned into a plottable x and y coordinate. This was achieved with some trigonometry.


```
def getCoords(self, measurement):
    x = measurement[1] * math.cos(math.radians(
        measurement[0]))
    y = measurement[1] * math.sin(math.radians(
        measurement[0]))
    return x, y
```

Each angle and distance pair is turned into its corresponding value on an x and y axis. Now that it was possible to read in data and turn it into a plottable point, it was time to combine this into an actual map.

```
# Draw map on plot
def showMap(self):
    measurements = self.readData()
    coords = []
    x = []
    y = []

    # Turn distance-angle pairs into usable
    # coordinates
    for measurement in measurements:
        x.append(self.getXCoord(measurement))
        y.append(self.getYCoord(measurement))

    plt.plot(x, y)
    plt.show()
```

This method was attached to a button. It uses the previously described `readData()` method to retrieve values from a file, turns each of the retrieved measurements into a plottable coordinate, appends it to an array and finally this array is plotted by the matplotlib scatterplot.

This was as far as this component got in development unfortunately. Issues with other features cost time, and problems were encountered with getting CSM to work. These problems are explained in 17.

14. TESTING

14.1 Introduction

This chapter will discuss the testing performed on the system. The aim of these tests is to ascertain how well, if at all, elements of the implemented work satisfy the project requirements outlined in 7. The tests themselves will be explain and justified, and a discussion of what conclusions can be drawn from the test results will follow. For a detailed break down of individual tests and their results, view the test logs in 20.1. Any code used to perform tests can be viewed in 20.2.

In keeping with the developmental methodology outlined in 11, tests were carried out at key functional milestones to ensure that what was implemented was robust and appropriate enough to be iterated on by the next prototype. When appropriate, it will be explained what stage of development the project was at during the test.

14.2 Integration Testing

Integration tests were performed to help determine the functionality of each of the system's individual elements, as well as the functionality of the system as a whole. These tests would help determine how well, if at all, certain aspects of functionality had been implemented as well as helping to detect runtime errors.

During the project's development a 'bottom up' style of testing was involved. This involved fundamental bottom level units being tested for basic functionality, and then further tests taking place as these units were joined together. This allowed for a basic implementation of each individual system element to be in place

This allowed for each specific part of the system to be tested to ensure that its implementation was proper before it was joined to other system elements.

14.2.1 Basic Movement Testing

One of the functional requirements of the project outlined in 7.1 is that robot must be able to move. To determine if basic movement was possible, a small test task was made within a Micro C OS environment. The task picked a random number from zero through to three, with each of the four numbers corresponding to one of four different movement methods that drove the robot in a different direction, the specifics of this are shown in table 14.1. The direction the robot was told to move in was printed to a terminal over a serial connection, and it was recorded whether or not the instructed location was the one the robot moved in. Relevant code can be seen at 20.2.1.

Number	Corresponding Direction
0	Forward
1	Right
2	Backward
3	Left

Tab. 14.1: Directions that each number represents

Test Results

Table 20.1 has the test's results in depth. Each of the appropriate directions printed to the terminal matched the direction that the robot traveled in. This means that the fundamentals behind how to manipulate the motors to achieve movement has been understood, and that the way in which this is done has been successfully implemented into a form usable by the robot's microcontroller. This only accounts for basic movement however, nothing can be surmised about more elaborate functionality involving true omnidirectional movement. As well, the test does not allow any insight into how well the wheels function once the chassis has more components and wiring in it.

14.2.2 LIDAR Testing

One of the project's functional requirements outlined in 7.1 is that the robot must be capable of observation. Initial LIDAR tests involved establishing whether or not a functional link could be made between the sensor and the microcontroller. A test task was made which began the sensor, outputted some data to a console, then stopped the sensor. This task can be seen at 20.2.2.

Test Results

A breakdown of specific results can be seen in 20.2. Each of the tests passed, indicating that the microcontroller is capable of starting and stopping the LIDAR sensor as well as retrieving data from it.

14.2.3 File Writing Tests

To obtain observational data so that it can be processed into a map, the robot must be capable of taking LIDAR data and writing it to the Micro SD-Card. To test whether this functionality was present, the test code at 20.2.3 was used. The code takes a series of LIDAR readings and prints them out to the console and also writes them to a created .txt file. As well as functionality, the code was then scaled up to see if a much larger number of readings could be written.

Test Results

Test results in 20.3. Each of the tests passed meaning functionality allowing the microcontroller to create and write to files was successfully implemented, and verification that the data matched what was printed on the terminal meant what was being written was what the LIDAR sensor was outputting. Once this was established, the for loop was increased to 8000 to see if a significantly larger number of readings could be written without any program issues. The system took longer to create the file, but still ran and ended without any issues and the produced file contained the additional readings. From this, we know it's possible for the microcontroller to receive and write the large quantity of readings that will be needed to produce an accurate map.

14.2.4 Mapping Tests

Tests shown in 20.4 were conducted to determine if the LIDAR sensor, microcontroller and produced GUI program could work as a module to produce a map. The same program used in the file writing tests was used to save a few thousand readings to a .txt file that was written to the Micro-SD card, which was then plugged into a PC with a USB adapter where the GUI was ran to generate a map.

Test Results

The results at 20.4 show that, whilst the GUI is capable of the preliminary functions of starting up and actually parsing observational data from a file, it doesn't process the data in a way that allows for an accurate map to be generated.

14.3 System Testing

As well as each individual component being tested, it was important that the system as a whole be put through tests to ultimately determine whether it had satisfied the initial project requirements.

With regards to the tests themselves, tests were created based on the requirements outlined in 7 as well as aspects like system robustness. Tests pertaining to things like error handling would invoke improper conditions and actions to see how the system dealt with edge cases, and tests pertaining to functionality would determine how well - if at all - the requirement had been implemented.

It's likely that these issues will be due to an improper robot state such as deadlock or starvation, or possibly a lack of memory owing to poor memory management within the microcontroller running the program. Runtime issues with the GUI are likely to be IO problems pertaining to issues reading the file containing the scan data as well as problems processing it into usable coordinates.

To begin with the robot's program was compiled and flashed onto the microcontroller. A fresh compile allowed for any compile time errors to be detected, and also meant it was certain that the program being tested was representative of the most current system state.

14.3.1 Test Results

Table 20.5 contains the specific test breakdown. The tests that passed mainly pertained to the gathering of observational data, with only one of the movement tests passing and the rest failing. The robot demonstrated the ability to scan data and write it to a Micro-SD card within a reasonable time frame. The successful scanning meant that the functional requirement of the robot needing to be able to perform observation was successfully met by the system, and the successful writing of the scanned data to the SD-Card within a reasonable time frame also satisfied an element of the robot's appropriate operational speed.

The drive forward task passing at least indicates that the motors are functional, but the functional requirement of the robot being capable of movement has an implication of it being multi directional. The other movement tests failing means that the requirement for the robot to move, whilst satisfied in a literal sense, has ultimately not been met.

The other tests that failed all pertained to the robot's capability to map areas. Whilst the GUI produced plotted points the maps did not resemble the environments that the robot was tested in. Based on this, it can be concluded that the functional requirement for the system to process scanned data into a map has not been satisfied. This has the follow on effect of the requirements for maps being accurate for different areas as well as consistent for the same area not being implemented as well.

On the whole, the system has not fulfilled the outlined project requirements.

Part III

EVALUATION

15. INTRODUCTION

The Evaluation chapter aims to take a critical look at the product and how well it satisfied the requirements. This will involve reviewing all hardware and software aspects of the system and discussing their strengths and weaknesses, with appropriate evidence to back up these discussions.

Following this will be an evaluation of the process that was undertaken to create the product. This will entail a discussion of any positives such as skills acquired and lessons learned as well as negatives such as performance problems. There will also be a discussion of potential alternative project plans using the benefit of hindsight to determine what might be done differently a second time around for the project to have enjoyed more success.

Finally there will be conclusions and recommendations. Project aims will be compared against work carried out, and conclusions will be drawn summing up what has been done and what it means. Following this, recommendations for further work will be discussed in the context of the project's wider scope.

16. PRODUCT EVALUATION

This chapter aims to be an evaluation of the product, comparing its functionality against the initial product requirements. Each of the requirements will be listed in turn, with a discussion of what evidence has been used to surmise how well it did or didn't meet the requirement, followed by a conclusion on whether or not the requirement was ultimately satisfied. The primary requirements identified for the product can be found in 7.

16.0.1 Functional Requirements

The first of the requirements is that the developed robot must be capable of movement. Tests performed in the synthesis pertaining to movement had some initial success with the robot being capable of moving in basic directions when the commands to do so were hard coded into the microcontroller's program, but the system tests showed that this movement did not serve much practical purpose during the robot's operation. It was not capable of dynamically changing its movement, only of travelling in whatever direction was hard coded into it. What was initially envisioned was a more dynamic and universal method of movement, with the robot changing its direction during operation for things like obstacle avoidance. It was also envisioned that this movement wouldn't be simple directions, but a complete omnidirectional movement capability. Instead of employing a few different methods for basic movement, the robot should have taken heading and distances to attempt to calculate the correct force that would need to be exerted on each wheel in order to drive it in the appropriate direction in the appropriate speed. Alternatively it could have used a simpler method of movement like the one implemented by Borenstein and Koren[7] for a simple nursing robot, where the only movement done is either a pure rotation or moving forward in the direction its facing. As well, the movement should have been dynamic and consistently dictated by the main program to steer the robot out of the way of obstacles. Using the robot's observation capacity, distance readings from the LIDAR sensor could have been used to determine if obstacles were close to the robot. If they were, the robot should have detected this and immediately changed its heading to navigate around them. The primary justification behind these pieces of functionality can be seen in the mobile robotics section of the analysis, where it is repeatedly mentioned

that movement and range finding were used for obstacle avoidance in a few different projects. For these reasons, it is believed this requirement has not been satisfied.

The second of the functional requirements is that the robot must be capable of observation. Initial integration testing for the observational capability showed promise, the microcontroller proved capable in controlling basic sensor functions as it was able to start, stop and retrieve data from the LIDAR sensor. This functionality didn't prove to be less effective during the system test either, as during the system test it still demonstrated the ability to take readings and store them to the Micro-SD Card medium. For these reasons, it is believed that this requirement was satisfied by the product. Despite this, the implementation could have been a bit more sophisticated. The returned readings data features an attribute called quality which indicates the validity and accuracy of the reading. Before being written to the buffer, this attribute should have been checked with low quality readings being discarded. This would mean only readings the sensor is confident are correct are written to the file and in turn, would mean the generated map would likely be more accurate. As well, the LIDAR supports a 'get health' request where it returns information pertaining to its current state, indicating if there is a warning or an error. This could have been called before the LIDAR sensor began its scanning to ensure right at the start of the robot's operation that the sensor was in a good state to begin outputting data.

The final functional requirement was that the observational data must be processed into a map. The actual GUI that was to process the observational data was produced and during testing demonstrated that it was capable of reading observational data in the format that the LIDAR and microcontroller's file writer system produced it in. When tests observed whether or not the produced map was accurate to the environment the readings were taken in however, it was immediately apparent that the readings were not properly processed. As well, even if this did work the readings should have been processed in a more efficient way. The best alternative to this simplistic mapping was the use of the CSM software discussed in the analysis, as this would have provided a much more sophisticated map and could have incorporated localization data from the robot as well. For these reasons, it is believed that this requirement was not satisfied.

16.0.2 Non-Functional Requirements

The first non-functional requirement of build quality concerned the physical soundness of the robot's construction. Examples of this not being satisfied are things like wired snagging the motors or LIDAR sensor, and components falling out of the structure during operation. System testing didn't observe

anything like this happening, and the structure remained intact. This was partially thanks to the use of tools like zip ties to secure wiring during the chassis construction. For these reasons, it is believed this non-functional requirement has been met.

The second non-functional requirement concerns the robot's operational speed. Key aspects highlighted in the requirement's explanation was that the scanning session should take less than ten seconds, and the map generation less than thirty. Some success was had here, with the robot able to obtain several thousand samples in under the given time ensuring that it operated at an appropriate speed. Unfortunately it was unable to be determined whether or not the GUI could generate a map in a sufficient time frame because the map it did produce was completely inaccurate. Whilst it produced this in a reasonable time, this could simply be because it was processed wrong and processing it right could involve a map generation that takes much longer. It therefore cannot be concluded that the system has a satisfactory operational speed, and as such this requirement has not been satisfied.

The third and fourth non-functional requirements concern the mapping procedure. As mentioned previously in the operational speed discussion and shown during the mapping integration tests and the system tests, the produced system is incapable of producing a map that represents the scanned environment. Given it's inability to fulfill the base requirement of producing a map, the requirements for this mapping to be adaptable and consistent have not been satisfied.

16.0.3 Conclusions

The majority of requirements outlined in the analysis have not been satisfied by the produced work, and this was primarily reflected by the numerous failed tests of the systems test conducted in the synthesis' testing stage. On the whole it is believed that the product failed to satisfy the original project aims of creating a self navigational drone, and using it for SLAM purposes.

17. PROCESS EVALUATION

The purpose of the process evaluation is to take a critical standpoint on how the project as a whole was approached. This will concern aspects such as how well the project was managed, what has been learned from the project during its undertaking and what would be done differently a second time around now that there is the benefit of hindsight.

17.1 Objectives

Most of the objectives outlined in the Terms of Reference have been met. Knowledge of embedded systems has certainly been enhanced thanks to project aspects such as making use of embedded documentation for different pieces of hardware, utilising embedded libraries to attain different pieces of functionality, and in particular connecting and controlling peripherals (e.g. the LIDAR sensor) using an embedded system.

Knowledge of LIDAR and SLAM has also been attained from the literature review. The literature review for mobile robotics contained a section on LIDAR, and good number of different resources were used to understand how LIDAR worked, the strengths and weaknesses of the technology, and also some of the different projects that have employed LIDAR and why they have chosen to do so. The same goes for SLAM, as the fundamentals of the SLAM problems have been properly reviewed and understood in its respective investigation. As well, a greater understanding of SLAM's usage in the wider world has been attained. Thanks to this investigation the viability of how SLAM might be achieved with the drone was understood better as well. Based on what knowledge was achieved by SLAM, an appropriate investigation into how the data might be process was conducted and was met with a conclusion that helped define a key element of the overall system.

Due to time constraints and developmental problems however the drone's construction was never really completed. Physically, the goal of construction was met given the assembly of the chassis and the fitting of some relevant components. Software-wise however the drone cannot be considered complete as its functionality is incredibly basic, lacking aspects such as obstacle avoidance and dynamic movement. As well, the inability of the created sys-

tem to perform SLAM lends itself further to the argument that construction hasn't been achieved. Before this objective can be met, a more thorough implementation of movement should be realised as well as a proper implementation of the system's mapping capabilities.

With regards to a proper evaluation of the product against the relevant aims and objectives, that can be called a success given the previously chapter discussing precisely this. The objective of discussing and evaluating how the project could be taken further is to follow at the end of the Evaluation.

17.2 Skills

The first skill outlined in the Terms of Reference is for the improvement of basic electronics assembly capability. This has been satisfied, proved by a physical implementation of the drone being mostly realised with an assembled chassis, microcontroller and LIDAR sensor all connected to each other. As well, the design featured a section discussing how the drone's power supply would be determined with consideration paid to voltages and battery capacity to calculate what sort of operational runtime could be expected from different power sources. Based on these things, it's concluded that this skill has definitely improved.

Embedded engineering skills were improved significantly as well. As previously discussed, the implementation of the robot's program on a microcontroller allowed for a greater appreciation of writing embedded software. As well, one aspect that was entirely new to the author was connecting different components to the microcontroller and manipulating them through the microcontroller's software. Establishing a connection between the LIDAR sensor and microcontroller and then being able to control the sensor through commands the microcontroller sent to it was new territory for the author, and the reasonable success of an observational functionality with the robot is proof that the understanding of this has been developed.

An improved understanding of \LaTeX has definitely been obtained from the project. Through the usage of online tutorials and resources the fundamentals behind creating a \LaTeX document were understood, and this allowed for the entirety of the project report itself to be written in \LaTeX . Not only has this allowed for a greater understanding of it as a technology, but an appreciation of the different features such as easier formatting, easier code insertion and the multitude of external libraries has been developed.

The skill that has probably made the least progress in the report is the knowledge of different SLAM algorithms. The fundamentals behind SLAM have been well researched and written about, and evidence of their understanding can be seen in the SLAM investigation in the analysis. The problem

with understanding the ins and outs of the different SLAM algorithms was how complex they were. Whilst the basics were understood, once the details of each of the algorithms such as the Extended Kalman Filter were discussed there was a vast gap of knowledge that was never successfully managed to be crossed. More preliminary research into SLAM might have helped to improve this skill.

17.3 Project Plan

Initially the project was approached well. Outlined in the project plan, the initial objectives during the first few weeks were to create the Project Initiation Document and the Terms of Reference, as well as to begin some reading into any relevant hardware documentation that had been established as potentially useful at the very beginning of the project. These were all achieved, and the initial objectives of the build phase involving creating some basic programs and hardware tests also went ahead fine. This gave some early confidence thanks to goal achievement, and the objectives were well designed as relevant project documentation was handed in on time and some preliminary knowledge of what might be used to create the project was garnered and understood. The Ethics form too was submitted on time and the marking scheme in the project handbook was reviewed, affording a good understanding of what the project report would entail. The project's initiation then was well planned which assisted in it being well executed.

Issues began to show themselves in November however, when the project's build objectives began to ramp up. The initial build tasks of creating a dummy board program and testing the LIDAR sensor being used for the project went ahead fine, and basic notes on the product's design were made in preparation for starting a draft on the design section. One task wasn't followed properly at this point, with the task of establishing communication between the microcontroller and the LIDAR sensor being attempted before the construction of the physical chassis. This was attempted first due to some overconfidence attained from previous work with the hardware, and a big problem was very quickly encountered. The previously studied SDK's documentation states that the `rplidar.h` file simply needs to be included to gain LIDAR functionality, but when this was done the program gave file dependency errors during compile attempts. The most prominent error was an import problem where files within the SDK were unable to import the file `sys/ioctl.h`. Looking into this, `ioctl.h` appeared to be a linux kernel header file. The first attempt to fix this was simply making sure these kernel files were working properly, so APT was used to install or update the linux kernel headers. Other packages such as build-essentials were also installed, but none of these things fixed the problem. One other potential problem

was that maybe there was a program somewhere relying on 32bit libraries which the 64bit Ubuntu OS didn't have, so APT was used to install the i386 architecture as well as some relevant packages to allow for better 32bit support. This didn't fix anything either. In a last ditch effort some of the kernel header files were actually copied into the LIDAR SDK, but this just resulted in other dependency problems coming up. In the Project Plan this task was estimated at taking around two weeks to complete, but at this point a month had elapsed and there was still no progress. In response to these failures the approach of using the SDK was abandoned, and an attempt to manually incorporate the protocol outlined in the observational section of the design chapter was attempted. Data structures of bytes were made and sent to the LIDAR through the serial connection byte by byte with a similar approach being used to retrieve the data it sent back. First attempts at doing this were unsuccessful, and it was a while before the LIDAR began sending something back. Interrupt methods that retrieved characters from the serial channel were configured, triggered by the presence of a character on the RxBuffer. Once this was done, data began being received at a steady rate. After this bytes began being stored into structs representing response descriptors and the structure of the responses themselves. Then the response data that was being retrieved from the LIDAR was converted to binary through the use of a byte lookup table that changed a given byte into its binary representation. This was due to how each bit in each byte had a specific purpose, an example of this can be seen in fig 12.8 in the design chapter. This still didn't really work, because most of the requests sent to the LIDAR didn't receive valid responses. For example the protocol documentation stated that the get health request returned three bytes but whenever it was tested it only returned two. As well, a request to stop the scanning procedure would immediately stop all data output and any subsequent attempts to start the scanning process again wouldn't work, requiring a hard reset of the board. By this point it was late December. Attempting to get the two components set up had consumed a lot of time, and the project report and product build had fallen behind massively. The analysis draft was only half finished and no physical aspect of the robot had been realized. Getting the two components communicating was completely abandoned as a task, and previous tasks involving creating drafts of report sections and assembling the robot's chassis were addressed. In hindsight, the decision to do this really should have been taken sooner.

Moving into the second semester, the lack of progress with a robot created the discussion of how a map might be created and displayed to a user. This discussion resulted in the goal of a GUI being created to process and display observational data to the user. The decision of this goal at this point in the project led to it not being included in the Project Plan, which further worsened the planning issues. Regardless, before February was over a basic

GUI had been made that showed a matplotlib instance with some dummy points plotted and the robot's chassis had been assembled with a battery pack power source on the way. As the semester went on, the massively behind analysis was worked on to try and create a draft and work was done to attempt to implement the robot's movement functionality. By March, a drone capable of basic movement had been created, meaning November's objective of having an automated drone was massively overdue. Partially this was due to developmental problems, partially it was due to mis-allocated time as what time had been spent on the project was wasted attempting to fix problems with the LIDAR sensor, and partially this was due to the incredibly over optimistic timescale of the plan's objectives. Looking back, achieving an automated drone in the space of a few weeks with just 30 hours of work is an incredibly naive goal, and far more time should have been dedicated to it. As well, 'automate drone' is a very vague and general objective, and should have been broken down into tasks of a more granular nature. This would have allowed for a clearer understanding of what needed to be done, a better time estimation, and an easier time meeting goals which would have helped boost confidence and morale during the project.

Eventually a solution to the sensor problem was stumbled upon in the form of a modified SDK from an open source project that used the LIDAR sensor with an arduino based robot, and communication between the LIDAR sensor and the microcontroller was established. By this point in the report, there was only an analysis draft and a rough draft of the synthesis' design and implementation sections, and a lack of product and report progress incited some panic and caused a few unsound decisions to be made. At the start of the product's development, a skeleton OS comprising of just one file was created. From this, different bits of functionality were tested and hacked in to get them working. For example with the movement, the control class discussed in the design wasn't implemented and was simply put in the main program as getting it working was the priority. As more and more code like this was added to get different things working, this initial prototype main file formed the basis of what little functionality had been implemented. When attempts were made to break it down into classes, memory issues were encountered and the already massively constrained schedule meant this problem couldn't be rectified, resulting in the robot's main program not resembling what was outlined in the design chapter at all.

This entire mismanagement of time, underestimation of task complexity and overestimation of personal skill manifested itself the most in the testing. Because of development time being so overstretched, far less time was left for thorough testing. Unit tests couldn't be conducted on time, and the bulk of the testing was the integration tests performed during the robot's development. These integration tests, whilst useful, were improper given the project's methodology however. The project methodology outlined in the

design was a prototyping style, aiming to first develop a high level prototype and then fill in functionality. The bottom up style of integration testing does not lend itself well to this approach as it focuses on modular functionality. A better approach for testing would have been a top down style, where all of the modules are attempted to be integrated very roughly and only after this is established is more specific functionality worked out. This would have resulted in a quicker prototype, an earlier achievement of the robot chassis construction and less time spent working out low level specifics which ate up the time of later objectives.

On the whole, the primary issues with the process were as follows. First was a vast underestimation of task complexity, more time should have been given to tasks to accommodate for problems. Second was too much tunnel visioning on tasks. After a few weeks of failure with the LIDAR, it should have immediately been abandoned and focus diverted elsewhere to prevent the project from falling even further behind. Third was poorly planned testing. Testing should have been given greater thought earlier on in the project to avoid a testing style that clashed so violently with the outlined methodology. In the future, time taken to deal with task failures should be thought about a lot more when deciding on task time frames. As well, a greater awareness of the wider project should be kept in mind to help prevent getting bogged down in very difficult problems and allowing them to consume too much development time, as well as to help ensure aspects like methodologies and approaches to different things like testing mesh well together.

17.4 Choice of Equipment and Techniques

This section aims to explore the strengths and weaknesses of the tools and techniques employed during the project's undertaking. This will primarily be achieved by looking at what the original reasons were for employing a tool or technique, and comparing these reasons with how the tool or technique served the project during development.

The majority of tools employed during the project's development were appropriate for their purpose. The three wheeled chassis provided a stable, easy to assemble platform to house the components of the robot. Whilst the dead reckoning aspect discussed in the techniques was never realised, the initial planning for it to possibly be incorporated into the robot's function was still a good idea and the encoders that came on the chassis motors would have helped with this. As well, even though omnidirectional movement was never implemented, the wheels and layout of them lent themselves well to supporting universal movement.

The lack of accurate map generation means it is unclear whether or not the sensor was a good choice. The myriad of issues encountered attempting to have it interface with the microcontroller were likely due to a lack of research prior to its usage rather than any fault with the design itself, so it wouldn't be fair to say that this means it was not an appropriate choice. Initial reasoning behind it a high sample rate and an SDK to afford it accuracy and premade functional implementation was sound however.

As mentioned before, memory issues plagued the software aspect of the robot when it was being broken down into the classes outlined in the class diagram. Mbed OS appears to feature relatively user friendly ways of being able to load programs onto the board and have their memory usage tracked. Micro C possess similar features but they aren't quite as elegant, and inexperience with hardware led to this being a huge blocker for implementing the software's proposed class design. This problem never really went away and resulted in the final program being only a single class, a far cry from the intended functional decomposition that could be been achieved if the class diagram had been properly implemented. In hindsight, Mbed OS was likely a better choice owing to its better documentation and syntactic sugar. Usage of an IDE supporting breakpoints of the embedded code like Keil Microvision might have helped here as well. A microcontroller with more memory potentially could have been used here as well, but given the relatively high amount of memory available on the K64F its likely this memory problem was due to a design problem somewhere rather than an insufficient board.

18. CONCLUSIONS AND RECOMMENDATIONS

18.1 Conclusions

A degree of success have been enjoyed with regards to the project's aims and skills. As previously discussed, skills outlined in the Terms of Reference such as embedded engineering ability have been furthered during the project's planning and development. As well, small objectives pertaining to acquiring knowledge on fields such as SLAM and LIDAR have seen success, and at this point the project and its respective process has been evaluated against the project's aims and objectives.

The aim of the project however was to develop a self navigational drone that made use of LIDAR technology to perform Simultaneous Localization and Mapping, and these aims have ultimately not been met.

The existing implementation is a simple chassis composed of powered motors driven by a microcontroller, capable of only incredibly basic movement. The implementation possesses an observational capability, but due to the lack of a mapping functionality it is unknown whether or not the observations the robot takes and writes to a file are accurate.

BIBLIOGRAPHY

- [1] arm MBED. Frdm-k22f specifications, . URL: <https://os.mbed.com/platforms/FRDM-K22F/> (Accessed: 08 December 2018).
- [2] arm MBED. Frdm-k64f specifications, . URL: <https://os.mbed.com/platforms/FRDM-K64F/> (Accessed: 08 December 2018).
- [3] *An introduction to Arm Mbed OS 5*. arm MBED. Available at: <https://os.mbed.com/docs/mbed-os/v5.10/introduction/index.html> (Accessed 13 December 2018).
- [4] Jerome Barraquand and J-C Latombe. On nonholonomic mobile robots and optimal maneuvering. In *Proceedings. IEEE International Symposium on Intelligent Control 1989*, pages 340–347. IEEE, 1989.
- [5] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Sensor Fusion IV: Control Paradigms and Data Structures*, volume 1611, pages 586–607. International Society for Optics and Photonics, 1992.
- [6] Jose Luis Blanco. Iterative closest point (icp) and other matching algorithms, 2013. Available at: [https://www.mrpt.org/Iterative_Closest_Point_\(ICP\)_and_other_matching_algorithms](https://www.mrpt.org/Iterative_Closest_Point_(ICP)_and_other_matching_algorithms) (Accessed 12 December 2018).
- [7] Johann Borenstein and Yoram Koren. Obstacle avoidance with ultrasonic sensors. *IEEE Journal on Robotics and Automation*, 4(2):213–218, 1988.
- [8] Andrea Censi. An icp variant using a point-to-line metric. 2008.
- [9] Howie Choset and Keiji Nagatani. Topological simultaneous localization and mapping (slam): toward exact localization without explicit localization. *IEEE Transactions on robotics and automation*, 17(2): 125–137, 2001.
- [10] Christopher M Clark, Christopher S Olstad, Keith Buhagiar, and Timmy Gambin. Archaeology via underwater robots: Mapping and localization within maltese cistern systems. In *Control*,

- Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on*, pages 662–667. IEEE, 2008. Available at: https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?referer=https://scholar.google.co.uk/&httpsredir=1&article=1055&context=csse_fac (Accessed: 02 December 2018).
- [11] Quoc Khanh Dang and Young Soo Suh. Human-following robot using infrared camera. In *2011 11th International Conference on Control, Automation and Systems*, pages 1054–1058. IEEE, 2011.
 - [12] MWM Gamini Dissanayake, Paul Newman, Steve Clark, Hugh F Durrant-Whyte, and Michael Csorba. A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on robotics and automation*, 17(3):229–241, 2001.
 - [13] Yongtae Do and Jongman Kim. Infrared range sensor array for 3d sensing in robotic applications. *International Journal of Advanced Robotic Systems*, 10(4):193, 2013.
 - [14] Ralph O Dubayah and Jason B Drake. Lidar remote sensing for forestry. *Journal of Forestry*, 98(6):44–46, 2000.
 - [15] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: Part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006. Available at: <http://everobotics.org/pdf/SLAMTutorial.pdf> (Accessed 02 December 2018).
 - [16] Owen Edwards. nrf24l01p. URL: <https://os.mbed.com/users/Owen/code/nRF24L01P/> (Accessed 08 December 2018).
 - [17] Jean J Labrosse. *MicroC/OS-II: The Real Time Kernel*. CRC Press, 2002.
 - [18] Jean-Paul Laumond, Paul E Jacobs, Michel Taix, and Richard M Murray. A motion planner for nonholonomic mobile robots. *IEEE Transactions on Robotics and Automation*, 10(5):577–593, 1994.
 - [19] Tom B Lauwers, George A Kantor, and Ralph L Hollis. A dynamically stable single-wheeled mobile robot with inverse mouse-ball drive. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 2884–2889. IEEE, 2006.
 - [20] Geunho Lee and Nak Young Chong. Low-cost dual rotating infrared sensor for mobile robot swarm applications. *IEEE Transactions on Industrial Informatics*, 7(2):277–286, 2011.
 - [21] Kevin Lim, Paul Treitz, Michael Wulder, Benoît St-Onge, and Martin Flood. Lidar remote sensing of forest structure. *Progress in physical geography*, 27(1):88–106, 2003.

- [22] Raashid Malik and Huiming Yu. The infrared detector ring: obstacle detection for an autonomous mobile robot. In *[1992] Proceedings of the 35th Midwest Symposium on Circuits and Systems*, pages 76–79. IEEE, 1992.
- [23] mbed official. URL: https://os.mbed.com/users/mbed_official/code/SDFileSystem/ (Accessed 11 January 2019).
- [24] Richard M Murray and Sosale Shankara Sastry. Nonholonomic motion planning: Steering using sinusoids. *IEEE transactions on Automatic Control*, 38(5):700–716, 1993.
- [25] RobotShop. 3wd 48mm omni-directional triangle mobile robot chassis, . URL: <https://www.robotshop.com/uk/3wd-48mm-omni-directional-triangle-mobile-robot-chassis.html> (Accessed: 08 December 2018).
- [26] RobotShop. 4wd 58mm omni wheel arduino robot, . URL: <https://www.robotshop.com/uk/4wd-58mm-omni-wheel-arduino-robot.html> (Accessed: 08 December 2018).
- [27] RobotShop. Rplidar a1m8 retailer, . URL: <https://www.robotshop.com/uk/rplidar-a1m8-360-degree-laser-scanner-development-kit.html> (Accessed: 08 December 2018).
- [28] RS-Online. Arduino uno rev3, . URL: <https://uk.rs-online.com/web/p/processor-microcontroller-development-kits/7154081/> (Accessed: 08 December 2018).
- [29] RS-Online. Arduino uno rev3 data sheet, . URL: <https://docs-emea.rs-online.com/webdocs/0e8b/0900766b80e8ba21.pdf> (Accessed: 08 December 2018).
- [30] *RPLIDAR A2 - Introduction and Datasheet*. SLAMTEC. Available at: https://www.robotshop.com/media/files/pdf2/1d208_slamtec_rplidar_datasheet_a2m8_v1.1_en_2_.pdf (Accessed 14 December 2018).
- [31] SLAMTEC. Interface protocol and application notes, . URL: http://bucket.download.slamtec.com/b90ae0a89feba3756bc5aaa0654c296dc76ba3ff/LD108_SLAMTEC_rplidar_datasheet_A1M8_v2.2_en.pdf (Accessed 08 December 2018).
- [32] SLAMTEC. Interface protocol and application notes, . URL: <http://bucket.download.slamtec.com/>

- 4ad536379886737a0ca660d4cf650a896a7ea09f/LR001_SLAMTEC_rplidar_protocol_v2.1_en.pdf (Accessed 08 December 2018).
- [33] SLAMTEC. Rplidar sdk documentation, . URL: http://bucket.download.slamtec.com/351a5409ddfbba077ad11ec5071e97ba5bf2c5d0a/LR002_SLAMTEC_rplidar_sdk_v1.0_en.pdf1 (Accessed: 08 December 2018).
- [34] SLAMTEC. Rplidar alm8, . URL: <https://www.slamtec.com/en/Lidar/A1> (Accessed: 06 December 2018).
- [35] Cyrill Stachniss. Introduction to robot mapping, October 2013.
- [36] Sebastian Thrun and John J Leonard. Simultaneous localization and mapping. *Springer handbook of robotics*, pages 871–889, 2008.
- [37] Carnegie Mellon University. Nasa-funded robotic sub finds bottom of world’s deepest sinkhole. *ScienceDaily*, 2007. Available at: <https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20190000015.pdf> (Accessed 02 December 2018).
- [38] Emanuele Vespa, Nikolay Nikolov, Marius Grimm, Luigi Nardi, Paul HJ Kelly, and Stefan Leutenegger. Efficient octree-based volumetric slam supporting signed-distance and occupancy mapping. *IEEE Robotics and Automation Letters*, 3(2):1144–1151, 2018.
- [39] Keigo Watanabe. Control of an omnidirectional mobile robot. In *1998 Second International Conference. Knowledge-Based Intelligent Electronic Systems. Proceedings KES’98 (Cat. No. 98EX111)*, volume 1, pages 51–60. IEEE, 1998.

Part IV

APPENDICES

19. TERMS OF REFERENCE

CM065: Individual Project

Project Terms of Reference

Name: Liam Brand

ID : w15002158

Course: Computer Science

Project Type: General Computing

Project Title: LIDAR Drone

Supervisor: David Kendall

Second Marker: Chris Rook



**Northumbria
University**
NEWCASTLE

Contents

Background to Project	2
Where did the idea come from?	2
Why is the project of interest?	2
Proposed Work	3
Aims of Project.....	3
Objectives	3
Skills	4
Bibliography	5
Resources	5
Structure and Contents of Project Report	5
Report Plan	5
Report Appendices.....	6
Marking Scheme	6
Project Report.....	6
Product.....	6
Physical Drone.....	6
Drone Software	7
Project Plan – Schedule of Activities.....	8

Background to Project

The purpose of this project is to develop a drone that utilizes LIDAR (Light Detection And Ranging) technology for the purposes of self navigation and mapping. Primarily, this will be achieved using embedded engineering.

Where did the idea come from?

The suggestion initially arose during my time on placement at Nissan. During a brainstorming session with someone discussing potential ideas, they mentioned that the engineers are supposedly lacking an all encompassing map for the assembly line. Instead, they have lots of different maps of varying scales for various parts of the line. Some form of drone that could automatically move around and generate a definitive 3D map would be valuable, as the map would offer a clear and consistent reference for navigation to any part of the assembly line. It could also provide useful in helping see how viable the addition or removal of different machinery might be.

Why is the project of interest?

The project is of interest to me personally for a few reasons. The biggest one for me is the introduction of hardware into the project. Whilst I don't have any problems with pure software development, the introduction of tactile equipment into things makes it more interesting for me. There is a satisfaction in the immediate feedback of hardware (an LED coming on, a wheel moving, etc) that I don't get as much from seeing a piece of software compile. I'm looking forward to being

able to measure my progress by the appearance of function of the drone itself as I build and program it. LIDAR is something I've never used before either, so I'm looking forward to becoming familiar with some technology that I haven't made use of in the past.

To a wider audience, the project is of use due to the potential uses of the technology if the scope is expanded. Self-navigational drones within themselves are not new, but there are many situations where people are beginning to see the advantages in their use. A paper (Tang & Shao, 2015) from the Northeast Forestry University talks about the possible applications in unmanned drones in forests, about how they can be used to survey forests, support forest management or track wildfires. The paper also goes on to talk about how these things would be achievable with low material/operational costs and reduced risks to personnel.

These drones would also have uses underground. In Malaysia there is an intricate cave system known as the Gomantong Caves. Using (among other things) an autonomous drone and LIDAR technology, a team discusses (McFarlane, et al., 2013) how they were able to map this cave network and the overlaying land surface with 'unprecedented precision', also stating that LIDAR scanning provides additional data that can be used biological inventory and management studies.

Proposed Work

The project will start off with me becoming accustomed with the hardware that will be used to make the drone. Following this construction of the drone will be begin. This will encompass fitting the motor, the wheels, the sensor and the board and making sure that it all communicates as it should. The first major milestone will be once the drone has been built and is able to perform a degree of self navigation, which will be tested by simply having it move itself around a created environment, most likely a small space with some basic obstacles in it.

Following this we will begin to look into additional uses for the drone. There will be an investigation into SLAM (Simultaneous Localization and Mapping) algorithms and how viable it would be to implement them into the drone. What happens after this will be largely dependent on the results of the investigation. Depending on hardware limitations and a few other factors, we will either move down the route of generating a map that is stored on the drone and able to be retrieved later or we will look to fit some form of wireless transmitter that can transmit the map data back to a hypothetical base camp. Finding out which of these two options is the most viable is something that will become apparent once the drone becomes functional.

Aims of Project

- To develop a drone capable of self navigation using LIDAR technology.
- To use the self navigation drone for SLAM purposes.

Objectives

- Further enhance knowledge of embedded systems
- Construct the drone
- Carry out tasks to ensure the drone can navigate around a created space
- Learn about LIDAR technology through literature review
- Learn about SLAM through literature review
- Investigate viability of the drone for SLAM
 - Depending on the viability, we will either –
 - Map areas and store the data locally on the drone

- Wirelessly transmit map data
- Transmit a simple “I’m Okay” signal
- Evaluation of the project against the aims and objectives.
- Evaluation of how the project could be taken further

Skills

Skill/Knowledge Required	Purpose of Skill	Ways to improve skill
Basic Electronics Assembly	Construction of the drone will require some familiarity with the basics of electronics assembly. The most important part will be the actual construction, but I’ll also need to investigate some other things such as determining the drone’s power draw and finding a suitable power supply for it.	Online tutorials as well as schematics and user manuals for the components that I’ll be using.
Embedded Engineering	Programming of the drone will require embedded knowledge. I’ll need to be able to ensure the LIDAR data is processed in a logical way so that the drone knows how to function based off of it, changing things like the direction it is heading in and potentially the speed that it is moving at.	Embedded engineering module labs and lectures.
LaTeX	In the interest of creating a more academically suitable report, LaTeX will be used rather than Microsoft Word. After picking up the basics it should prove much easier for me to format chapters, citations and bibliographies among other things.	Online documentation.
SLAM Algorithms	For the drone to successfully implement a SLAM algorithm I will need to understand them that allows me to implement it. Knowledge of a few different algorithms would be preferable as it would allow me to implement the most suitable one.	Online reports and textbooks.

Bibliography

- McFarlane, D. A., Buchroithne, M., Lundberg, J., Petters, C., Roberts, W., & Van Rentergen, G. (2013). *Integrated Three-Dimensional Laser Scanning And Autonomous Drone Surface-Hotogrammetry at Gomantong Caves, Sabah, Malaysia*. Retrieved from https://www.researchgate.net/profile/Manfred_Buchroithner/publication/261097340_Gomantong_Congress_Paper_2013/links/00b7d5332d8cf3bd16000000.pdf
- Tang, L., & Shao, G. (2015, June 21). *Drone remote sensing for forestry research and practices*. Retrieved from https://www.researchgate.net/profile/Lina_Tang2/publication/283655699_Drone_remote_sensing_for_forestry_research_and_practices/links/5672302108aecc73dc09c776.pdf

Resources

3WD 48mm Omni-Directional Triangle Mobile Robot Chassis

FRDM-K22F MBed Board

K22F Reference Manual - <https://www.nxp.com/docs/en/reference-manual/K22P121M120SF7RM.pdf>

K22F User Guide - <https://www.nxp.com/docs/en/user-guide/FRDMK22FUG.pdf>

RPLidar A1M8 - 360 Degree Laser Scanner Development Kit

LIDAR Sensor Manual - https://www.robotshop.com/media/files/pdf2/ld108_slamtec_rplidar_datasheet_a1m8_v1.1_en_2_.pdf

Structure and Contents of Project Report

Report Plan

Table of Contents
Abstract
Introduction
Analysis
Problem Identification
Literature Review
Solutions
Product Requirements
Review of Tools and Techniques
Synthesis
Design
Implementation
Testing
Evaluation
Product Evaluation
Process Evaluation
Conclusion and Recommendations
Appendices

Report Appendices

Terms of Reference

Ethics Form

Software Documentation (e.g. test plan, use case)

Marking Scheme

This project is to be marked according to the General Computing Project marking scheme.

Project Report

Report Section	Relevant Marking Section
Abstract	Abstract
Introduction	Introduction
Analysis	
Problem Identification	Analysis
Literature Review	
Solutions	
Product Requirements	
Review of Tools and Techniques	
Synthesis	
Design	Synthesis
Implementation	
Testing	
Evaluation	
Product Evaluation	Evaluation of the Product
Process Evaluation	Evaluation of the Product Process
Conclusions and Recommendations	Conclusions and Recommendations

Product

The following section lists the deliverables of my project and the criteria which they will be assessed upon to determine their fitness for purpose, build quality or both.

Physical Drone

Fitness for Purpose

To be fit for purpose the drone must achieve one of the core project aims of developing a self-navigational drone that utilises LIDAR technology.

The way in which it achieves this must be of an acceptable quality as well. The drone must be able to detect obstacles and navigate around them before reaching them, if it is coming toward a wall it must stop and change direction before crashing, else the self-navigation is not sufficient.

The drone must also be able to perform SLAM functions. As it navigates itself around a space it should transmit map data back to a terminal.

Build Quality

To be of an acceptable build quality, the provided tests and their results should show that it has been thoroughly tested to check for faults that might occur in the maximum possible amount of situations that the drone would find itself in.

Drone Software

Fitness for Purpose

The primary way to determine whether the code is fit for purpose is to ensure that the drone is performing the necessary functions. It can be assumed from this that the code has been appropriately written to achieve what was needed. The language that the code is written in should be justified within the report however, with outlined specifics as to why the drone's software is written in was chosen.

Build Quality

The code itself should be written to a good standard, comments should be plentiful ensuring that it is clear what various parts of the code are achieving with appropriate function and variable names. As well, the code should be broken down into appropriate functions and methods with no large cumbersome super functions performing many different tasks at once.

Project Plan – Schedule of Activities

[illegible]

20. TESTING

20.1 Tests

20.1.1 Integration Tests

20.1.2 Movement Tests

20.1.3 LIDAR Tests

File Writing Tests

Mapping Tests

20.1.4 System Tests

Printed Direction	Direction Moved In	Test Result
Right	Right	Pass
Left	Left	Pass
Forward	Forward	Pass
Left	Left	Pass
Right	Right	Pass
Forward	Forward	Pass
Right	Right	Pass
Left	Left	Pass
Backward	Backward	Pass
Forward	Forward	Pass
Backward	Backward	Pass
Left	Left	Pass
Right	Right	Pass
Right	Right	Pass
Backward	Backward	Pass
Forward	Forward	Pass
Forward	Forward	Pass
Backward	Backward	Pass
Right	Right	Pass
Forward	Forward	Pass
Backward	Backward	Pass

Tab. 20.1: Movement Integration Tests

Test	Test Purpose	Expected Result	Actual Result	Pass/Fail
Begin sensor	Determine microcontroller's ability to start LIDAR sensor	Sensor starts spinning	As expected	Pass
Output reading	Determine ability to retrieve data from the LIDAR sensor	Terminal prints scan data	As expected	Pass
Stop sensor	Determine microcontroller's ability to stop LIDAR sensor	Sensor stops spinning	As expected	Pass

Tab. 20.2: LIDAR Integration Tests

Test	Test Purpose	Expected Result	Actual Result	Pass/Fail
File Creation	Ensure the robot can create files	File present on Micro SD-Card	As expected	Pass
Basic File Writing	Ensure created file contains data	File shouldn't be empty	As expected	Pass
Accurate Data	Ensure written data is what the LIDAR has produced	File's data should match what has been output on the terminal	As expected	Pass
Intense File Writing	See if the system can cope with writing many thousands of readings	File should contain more data, system task should still finish as normal	As expected	Pass

Tab. 20.3: File Writing Integration Tests

Test	Test Purpose	Expected Result	Actual Result	Pass/Fail
GUI Starts	Ensure the GUI starts properly	GUI will start up after being called from the command line	As expected	Pass
File Readings	Ensure the GUI can process the given file	GUI shouldn't encounter errors processing values from the file	As expected	Pass
Basic Map Generated	Ensure a basic map can be generated	GUI should display a map of the environment the robot mapped	Map was not accurate	Fail
CSM Map Generated	Ensure the GUI can use CSM to generate a map	GUI should output a map incorporating multiple scans	No such functionality implemented	Fail

Tab. 20.4: File Writing Integration Tests

Test	Test Purpose	Expected Result	Actual Result	Pass/Fail	Comments
Drive Forward	The robot should move forwards during its operation	As expected	The robot moved forwards when it was switched on	Pass	Hardcoded movement
Drive Right	The robot should move right during its operation	The robot is able to move right	The robot couldn't move right	Fail	Plugged in and viewed with USB adapter
Drive Left	The robot should move left during its operation	The robot is able to move left	The robot couldn't move left	Fail	
Obstacle Avoidance	The robot should navigate around obstacles during operation	The robot is able to avoid obstacles	The robot didn't detect obstacles	Fail	
Motor Obstructions	During the previous drive tests, there should be no obstructions to the motors from the robot's other components	As expected	The robot's motors didnt get blocked	Pass	
Scanning	The robot should stop to scan	The robot stops and scans its environment	As predicted	Pass	
Write Scan Data	Ensure the robot can write scanned data to the Micro SD-Card	The Micro SD-Card will contain a file with scan data	As expected	Pass	
Writing Duration	Ensure the robot writes data to the Micro-SD Card in a reasonable time	Writing should be done in under ten seconds	As expected	Pass	
Generate Map	Using the robot's scan data, the GUI should produce a basic map	Map will be produced resembling the robot's environment	Map was produced but didn't resemble the environment	Fail	
Generate Map Without File	Make the GUI attempt to generate a map without an available file	GUI will catch and handle a runtime exception	As expected	Pass	
Map Multiple Areas	Mapping should work on multiple different environments	Each map will be accurate to the environment the robot was in	Maps didn't resemble the environment	Fail	
Repeated Mapping	Repeated maps of the same area should look similar	Map will be produced resembling the robot's	Maps didn't resemble the environment	Fail	

20.2 Test Code

20.2.1 Basic Movement Testing

```
while (true) {  
    // Random number between 0 and 3  
    int r = rand() % 4;  
  
    if(r == 0) {  
        pc.printf("Forward");  
        goForward();  
    }  
  
    if(r == 1) {  
        pc.printf("Right");  
        goRight();  
    }  
  
    if(r == 2) {  
        pc.printf("Back");  
        goBackward();  
    }  
  
    if(r == 3) {  
        pc.printf("Left");  
        goLeft();  
    }  
  
    OSTimeDlyHMSM(0,0,3,0);  
}
```

20.2.2 LIDAR Testing

```
static void appTaskLidarTest(void *pdata) {  
    dtr = 0;  
    rplidar.begin(lidar_device);  
    rplidar.startScan();  
    struct RPLidarMeasurement measurement;  
  
    while (true) {  
        dtr = 1;  
        rplidar.waitPoint();  
        measurement = rplidar.getCurrentPoint();  
    }
```

```

    pc.printf("Distance: %f Angle:%f\n",
        measurement.angle, measurement.distance);
    dtr = 0;
    OSTimeDlyHMSM(0,0,1,0);
}
}

```

20.2.3 SDFFileSystem Testing

```

dtr = 1;
rplidar.begin(lidar_device);
rplidar.startScan();
struct RPLidarMeasurement measurement;

static void writeTest() {
    struct RPLidarMeasurement measurement;

    // Create the readings file on the Micro SD-
    Card
    FILE *fp = fopen("/sd/readings.txt", "w");

    // Log if the file cannot be made
    if (fp == NULL) {
        pc.printf("Unable to access/create file \n");
    }

    for(int i = 0; i < 10; i++) {
        lidar.waitPoint();
        measurement = lidar.getCurrentPoint();
        pc.printf("Angle:%f Distance:%f\n",
            measurement.angle, measurement.distance);
        fprintf(fp, "%f %f\r\n", measurement.angle,
            measurement.distance);
    }

    // Close file
    fclose(fp);
}

```