Construction of a SLAM Capable Robot

Liam Brand

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

ANALYSIS

1. INTRODUCTION

Before creation of the main product can begin, an understanding of the relevant fields must be achieved. The first aim of the analysis is to develop this understanding by evaluating and then understanding relevant literature. Following this appropriate tools for the product will be identified, both hardware and software. Finally conclusions will be drawn based on what has been researched and evaluated, and a series of core objectives that need to be achieved for the project to be a success will be determined.

2. PROBLEM IDENTIFICATION

3. MOBILE ROBOTICS

3.1 Introduction

Here we will address some of the key aspects of mobile robotics that are relevant to the project, chiefly how movement is dealt with in mobile robotics as well as range-finding.

3.2 Movement

This section aims to explore how movement is generally 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.

3.2.1 Chassis

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. This can present a few problems. Firstly, navigation will sometimes involve the adjustment of the vehicle's heading. The vehicle may need to reverse and turn before it can move forward to a certain location. This presents a problem in the robot's efficiency and ability to navigate around a difficult environment, as well as making it more challenging to implement from a software perspective as we would need to factor in situations where these sorts of adjustments would be needed before the robot can proceed. If the robot were to be holonomic however, then it would be able to begin travelling toward any location regardless of its current position. This can be achieved if the wheels our robot's chassis uses are omni-directional.

Watanabe[19], 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 omni-directional 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.

Fig. 3.1: Mecanum wheels with 90 and 45 degree rollers

A robot chassis composing of three or four of these wheels as well as relevant motors should be adequate.

3.3 Rangefinding

To achieve self navigation as well as mapping, the robot will need to take observations about its surrounding environment. In order to efficiently map the area and detect obstacles this will need to be performed in a 360 degree manner as well. This sections aims to explore a few of the most common approaches to rangefinding.

3.3.1 Infrared

Infrared 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 reciever will capture this reflected light and depending on factors such as how much light is recieved back and the triangulation of how the light was recieved the presence of an obstacle will be determined. There are a number of Infrared sensors available online for very cheap prices, websites like RobotShop and HobbyTronics list most of their sensors between Â£5 and Â£10. 360 sensors are incredible difficult to find, but their low price means purchasing a few would not be a problem.

The range and degrees of space being measured will depend on the sensor's model, but there are a few things the different sensors have in common. 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. In addition, infrared sensors struggle in strong sunlight and seem best suited to primarily indoor tasks.

3.3.2 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 has seen some popularity in mobile robotics and the decision to use LIDAR would give the project some interesting options. Some sensors boast very impressive statistics, with some ranges exceeding 10 metres whilst providing several thousand samples per second with 360 degree coverage [14]. 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. These features are expensive however, with some sensors being as high as £350.

3.3.3 Ultrasonic

Ultrasonic sensors function by sending out ultrasonic pulses and measuring the amount of time it takes for these pulses to bounce back. Because sonar is sound based rather than light based, it isn't negatively affected by aspects such as heat, colour or dust. Sensors can interfere with each other however, sonar sensors sending and receives pulses in close proximity to each other can cause issues. A single robot is only being developed, but given that we may need multiple sensors to perform a full 360 degrees of observation this could cause some considerable complications. Sonar sensors can be found relatively cheap depending on the sensor, with sites like HobbyTronics and RoboShop featuring sensors as cheap as £6 ranging up to some that cost around £50.

4. AN INVESTIGATION INTO SLAM

4.1 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.

4.2 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[8], 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 [5, 7].

4.3 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 [6] 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 [17] 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.

4.4 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 ith landmark within the environment.
$\mathbf{z_t}$ - Observation of ith 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 teh robot's movement.
$\mathbf{m} = \{ \mathbf{m_1}, \mathbf{m_2}, ..., \mathbf{m_n} \}$ - 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

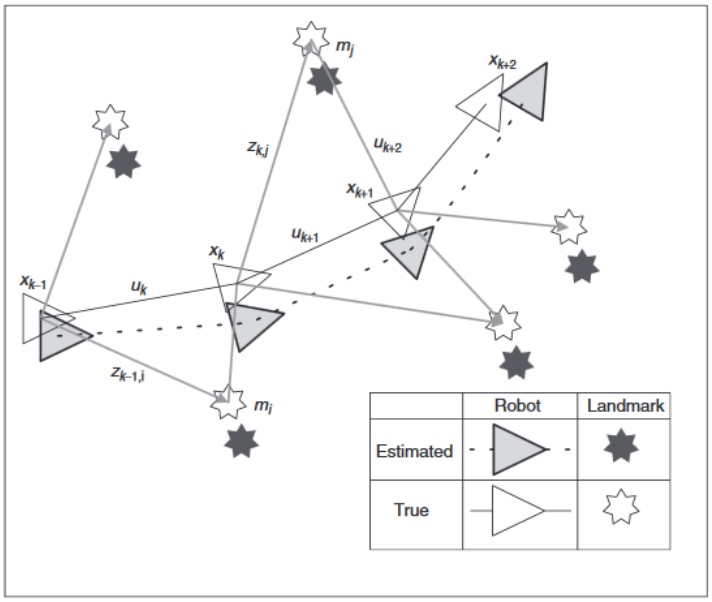

Fig. 4.1: The SLAM problem illustrated [8]

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.

4.4.1 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 -

$p(\mathbf{X_{0:t}}, \mathbf{m} \mid \mathbf{Z_{0:t}}, \mathbf{U_{0:t}})$

4.4.2 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 -

$p(\mathbf{x_{0:t}}, \mathbf{m} \mid \mathbf{Z_{0:t}}, \mathbf{U_{0:t}})$

4.4.3 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 [16]. 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 [18]. 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) [16]. 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.

4.5 A Look At A Potential Solution

4.5.1 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.

4.5.2 CSM

CSM is an open-source C implementation of an ICP variant known as PlICP. 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 PlICP 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 [2] in their paper, and ICP is shown in terms of C++ in the Mobile Robot Planning Toolkit [3].

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 a the mean squared error falls below a certain threshold.

PlICP

Censi explains PlICP in a series of steps [4]. 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 threshhold). 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. PlICP 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.

4.5.3 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.

4.6 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 revelations 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.

5. 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.

5.1 Functional

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

5.2 Non-Functional

- The robot's movement should be omnidirectional. To be fully capable of proper navigation in any environment, it would be ideal that the robot has full freedom of movement.
- The robot should be powered appropriately, with consideration given to battery and component voltages as well as power consumption.
- The observational data should be stored appropriately. As previously mentioned in the Terms of Reference, the way in which observational data is stored is to be determined via an investigation. The way in which data is stored should depend on the results of this investigation.
- Observational data will be processed into a map using the CSM software.

6. 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.

6.1 Tools

6.1.1 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

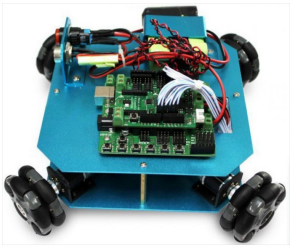

Fig. 6.1: 4WD Omni-Directional Robot Chassis

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 odemetric 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.

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

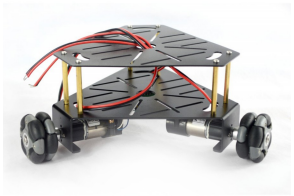

Fig. 6.2: 3WD Omni-Directional Robot Chassis

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.

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.

6.1.2 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 - £Â£20.80

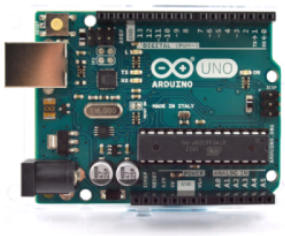

Fig. 6.3: Arduino Uno Rev3

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 RobotShop and HobbyTronics. Unfortunately the technical specification is somewhat underwhelming, as it only boasts 30 kB of available memory with an also slow clock speed of 16MHz. 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 - £24.45

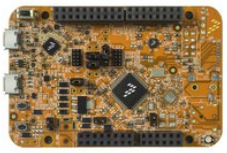

Fig. 6.4: FRDM-K22F

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 impres-

sive 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. 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. 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. 6.5: 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. 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.

6.1.3 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. Here we will look at a few different LIDAR sensors that could be suitable for the project.

Fig. 6.6: RPLIDAR A2M8

RPLIDAR A2M8 360 Degree Laser Scanner - £301.43

The RPLIDAR A2M8 uses the previous described LIDAR triangulation system, and outputs scan data at 8000 samples per second[12]. 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, 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. 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 at the highest sampling rate the sensor can use.

Fig. 6.7: RPLIDAR A1M8

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.

6.1.4 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[1] 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 recieving 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.

µC/OS-II

µC/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[10] 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.

6.2 Techniques

6.2.1 Storing Scan Data

One aspect that needed to be addressed was how the observational data was actually dealt with. In the Terms of Reference, a few different options were considered. These were either storing the data locally on the drone, wirelessly transmitting the map data or transmitting a simple signal back to a hypothetical base camp. What follows is a brief investigation into what the most viable approach might be.

Wireless Transmission

In order to either transmit the map data or any other signal, some form of transmitter will likely need to be added to the microcontroller. An RF (Radio Frequency) transciever could be added to the microcontroller allowing it to transmit and recieve radio signals. The K64F features headers for use with 2.4GHz radio add on modules, and a nRF24L01P Nordic transciever can be obtained very cheaply for just a few pounds online. The Mbed website also has a library[9] for this transceiver which would allow for interaction with it to be relatively straight forward. The problem here though is that having just a transceiver on the robot's microcontroller probably wouldn't be enough for interaction with it, a second microcontroller would probably

need to also have a radio chip soldered on so that something could actually receive transmissions. Attempting to get pairs of microcontroller listening to eachother presents problems in of itself, and as well the use of a radio would mean the robot's range is potentially limited.

Local Storage

The alternative 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. Similarly to the wireless radio transmission approach, mbed features an SDFileSystem library[11] 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. There's a possible problem with this though. 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 6.1 shows the results of these tests.

Test	Time in Seconds
1	7.9
2	7.9
3	7.7
4	7.7

Tab. 6.1: SD file write speed tests

Writing several thousand samples to the SD-Card doesn't seem to 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. These factors combined with the readily available components make local storage the choice for holding the

map data.

6.2.2 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 pressingly 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 [5]. 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.

7. CONCLUSIONS

Based on the different aspects of mobile robotics and SLAM that have been looked at as well as the critique of the various tools and techniques that have been employed, we can outline the basic product requirements that will need to be achieved in the course of the product's development. First and foremost the robot must be capable of basic navigation. The drone must be capable of observing the environment using rangefinding techniques. Finally, the robot must be capable of processing data acquired through this rangefinding process to perform localization and mapping. Finally, the retrieved observational data must be processed and fed into the CSM software to generate a map. For this to happen the robot needs to be able to somehow store the observational data it retrieves from the LIDAR sensor, and we also need a medium to put the data into CSM and display the output. From this outline we can get these basic project requirements -

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

Part II

SYNTHESIS

8. 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 established the methological 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.

9. METHOLOGICAL 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.

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.

9.0.1 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 eachother. 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 threshhold.

10. DESIGN

10.1 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.

10.2 Initial Prototype

Here will be the design for the base product that will serve as a foundation for all subsequent the development. The aim here is to have a skeleton project that can be filled in with functionality and built on to meet the project objectives.

10.2.1 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 10.1 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

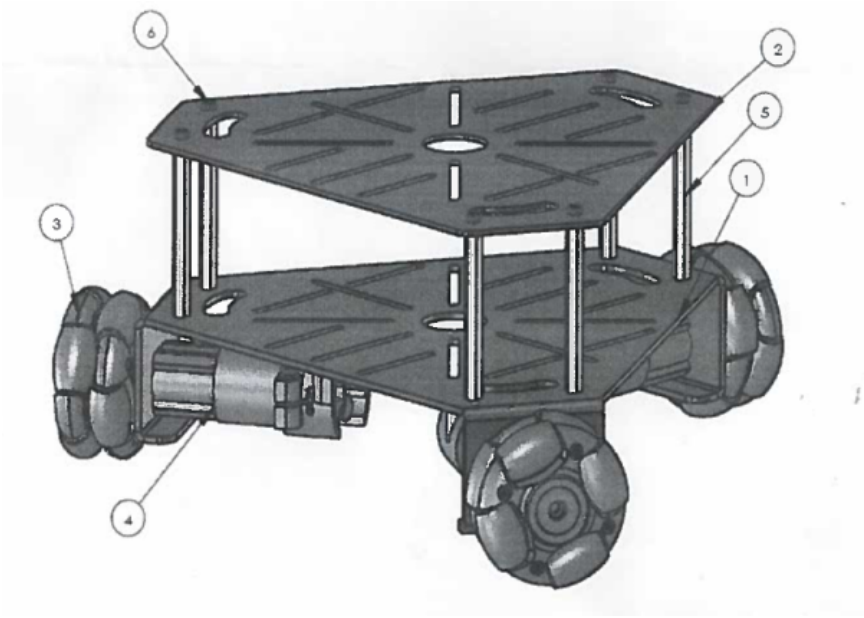

Fig. 10.1: Chassis assembly guide

Motor Driver Shield is being employed for this project, it can be obtained 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 preperation 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.

10.2.2 Software

Robot Program

As previously mentioned in the analysis, the robot's program will be composed of a Micro C OS. Once further design has been discussed and a good idea of how the program should be structured has been understood, it would be prudent to create a skeleton OS with some basic tasks that simply print to ensure first that software can be properly deployed onto the microcontroller, and secondly to speed up development by allowing individual parts of the OS to be developed straight away.

10.2.3 Power

The hardware components that will need powering are the three motors used to drive the omnidirectional wheels, the microcontroller, and the LIDAR sensor. Table 10.1 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. 10.1: 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. Fig **??** has an overview of this.

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.

10.2.4 Software

The software for the actual robot will be written in C++, compiled and deployed onto the microcontroller with the mbed SDK. At its foundation, it will be a Micro C OS with a series of configured tasks to perform various aspects of the robot's functionality. As would be expected, these tasks will come with their own memory tasks and will be designated with appropriate priorities.

The plan is for there to be three separate tasks. One task will deal with the robot's movement, one task will deal with taking data from the LIDAR and the final task will deal with actually writing this scan data to the connected Micro SD-Card. To control the program flow semaphores will be employed. For the initial prototype, these three tasks will be initialized and simply print messages to the console so a program such as minicom can see them running. This will allow assurance that the tasks are running in a logical order.

10.3 Designing for Requirements

Once this incredibly basic initial prototype is in place, 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.

10.3.1 Movement

The first of the three product aims outlined in the Analysis is that 'the robot must be capable of movement'.

Hardware

The power and ground cables from each of the chassis motors will be plugged into the motor driver shield. Fig 10.2 from the shield's documentation shows an example of this setup with four motors.

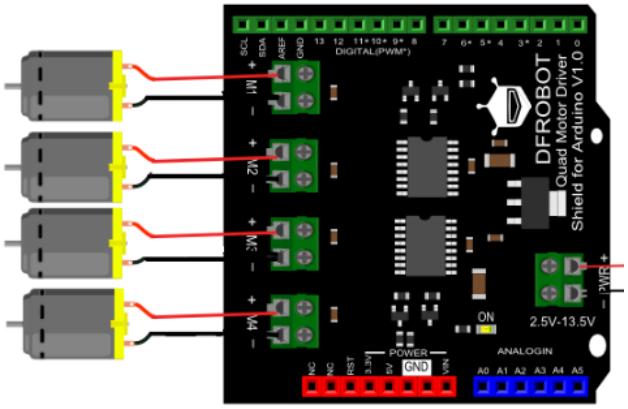

Fig. 10.2: Example shield setup

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

In order to manipulate the motors we'll need some appropriate variables we can use to refer to the pins that the motor power and ground cables are connected to.

10.3.2 Observation

The second of the three product aims outlined in the Analysis 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 10.3 illustrates these components.

Fig. 10.3: Example shield setup

There are seven pins on the underside of the LIDAR sensor, shown in fig 10.4. 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 need

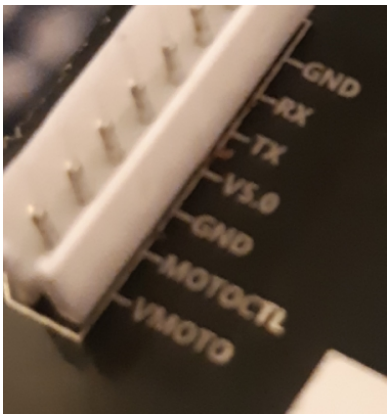

Fig. 10.4: LIDAR Underside

to be connected to pins on the microcontroller that output the appropriate amount of voltage. 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 recieve 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.

To interact with the sensor, it will be connected to the robot's microcontroller. We would first need to connect the microcontroller's TX (transmit) and RX (recieve) pins to the LIDAR's inverse pins (TX to RX, RX to TX). Then, using mbed's serial library, we can create a serial variable supplying the microcontroller's connected RX and TX pins and parameters. From this we can invoke methods like putc() and getc(), which put and get characters into and from the serial connection respectively. In order to power the sensor, there are two pins that need to be connected. The first is LIDAR core, which is the power for the actual LIDAR scanner itself. The second is the LIDAR motor, which is what spins the scanner as it makes observations.

Software

SLAMTEC have a document detailing the LIDAR protocol. This details the specifics behind how to communicate with the LIDAR. LIDAR communication is primarily achieved through the exchange of data packets. As a basic example, in order to obtain scan measurements the host system first needs to send data packets corresponding to the begin scan command. Once the LIDAR has recieved and processed this, it should begin sending back observational data.

The format of the requests that need to be sent is documented in the LI-

DAR's protocol documentation[13]. To implement this, it would be a good idea to create a data structure in the microcontroller's C++ program with fields listed in the protocol (start flag, command, etc) and then put this field into the serial connection. A similar data structure could be populated with what is recieved from the serial channel to make it easy to process the data.

SLAMTEC provide an SDK for the LIDAR sensor however, which comes in the form of header files which will automatically implement this functionality. Table 10.2 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. 10.2: RPLIDAR SDK files

Essentially, the C++ program running on the microcontroller would need 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.

10.3.3 Processing Observational Data

The last of the three project objectives 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 that after a session of moving and scanning 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. The Mbed microcontroller have 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.

Once readings have been written to the SD-Card, it will be removed and plugged into a USB adapter so it can be plugged into a computer. For ease of use, a GUI program will be used to automate the rest of the functionality. 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. The GUI's job then 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.

10.4 Pseudo Code

To help illustrate the system's overall design some pseudo code has been made. To prevent it from being overcomplicated some aspects such as the declaration of task priorities or memory stacks have been omitted, the pseudo code's purpose is not to show each and every aspect of what needs to be implemented but rather to serve as a structural guide during the implementation.

```
FileSystemDeclaration
LIDARSerialConnectionDeclaration

scanningSemaphore
writingSemaphore

movementTask() {
 acquireSemaphores
 move
 releaseScanSemaphore
 pendScanSemaphore
 releaseWritingSemaphore
 pendWritingSemaphore
}

scanTask() {
 pendScanSemaphore
 scan
 releaseScanSemaphore
}

writeTask() {
 pendWritingSemaphore
 write
 releaseWritingSemaphore
}
```

11. IMPLEMENTATION

11.1 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.

11.2 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.

11.2.1 Hardware

First of all the physical chassis had to be assembled. The only available documentation was fig 10.1, a labeled picture of an already assembled chassis which wasn't terribly useful. Luckily the construction was relatively straight forward, and only took around thirty minutes. Once the structure 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.

11.2.2 Software

To speed up later development a basic Micro C OS was created and deployed onto the microcontroller. This OS resembled the pseudo code and was composed of skeleton tasks that simply printed their task name to the console when they ran. Semaphores were initialized and combined with the print messages the basic program flow could be figured out.

11.3 Initial Prototype

11.3.1 Hardware

11.3.2 Software

11.4 Implementing Requirements

11.4.1 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 hereonn 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 recieve power from the microcontroller. The second decision was that the microcontroller would receive its own battery.

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 span 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.

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

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

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

Simple movement could be achieved just by configuring different motors to spin in different directions initially. Functions were made that allowed the robot to drive in four basic directions. One such function can be seen below.

```
static void goForward() {
 pc.printf("goForward");
 M2_DIR = 1;
 M3_DIR = 1;

 M1_SPD = 0;
 M2_SPD = 0.5f;
 M3_SPD = 0.5f;
}
```

This for example set Motor 2 to spin clockwise and Motor 3 to spin counter-clockwise. Both of these wheels therefore moved in the direction of the first motor, and the small rollers on the wheel allowed the robot to frictionlessly slide in this direction.

11.4.2 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 recieving 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. 11.1: RPLIDAR SDK files

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 recieve 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 recieve 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 recieve 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.

To first of all ensure the LIDAR is ready to communicate, the previously discussed MOTOCTL pin on the LIDAR needs to be recieving a HIGH signal so that it knows the microcontroller is ready to begin recieving data. The K64F features numerous GPIO pins that can be easily configured for usage in situations like this. One such pin (D7) was simply declared as a basic DigitalOut allowing the microcontroller to manipulate the pin's polarity.

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

When we want the sensor to output data this pin can be set to HIGH...

```
dtr = 1;
```

...or when we want it to stop, we can set it to LOW.

```
dtr = 0;
```

Once this was done and the DTR pin was set to HIGH, the LIDAR began spinning. The microcontroller didn't instantly begin recieving data 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.

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

As previously discussed in the Design, 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 main.cpp file.

```
RPLidar lidar;
```

Following this a base lidar variable can be declared from the RPLidar class the SDK adds.

```
RPLidar lidar;
```

Once this variable has been declared, in the program's 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.

```
dtr = 0;
lidar.begin(lidar_device);
lidar.startScan();
```

From here, two different simple one-line methods are used to start and stop the lidar.

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

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

These methods allow for quick and easy configuring of the DTR pin.

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 recieving reading, the RPLidar SDK implements a basic struct that can be filled in with a .getCurrentPoint() method. This will be used so that recieved readings can be easily manipulated.

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

Now we need somewhere to store the readings themselves so they can be written to disk later on. We're interested in storing a large number of angle/distance pairs, so a large two dimensional array was used.

```
float readingsBuffer[16000][2];
```

A two dimensional array was used to ensure an understandable format to the array. The array has 16000 spaces, with each space storing two seperate float values (angle and distance respectively). Now the actual method to take LIDAR readings and store them in the buffer needs to be made.

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

```
static void takeReadings() {
 int arraySize = (sizeof(readingsBuffer)/sizeof(
    float))/2;
 struct RPLidarMeasurement measurement;
 for(int i = 0; i < arraySize; i++) {
  lidar.waitPoint();
  measurement = lidar.getCurrentPoint();
  readingsBuffer[i][0] = measurement.angle;
  readingsBuffer[i][1] = measurement.distance;
 }
}
```

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

All of these methods can now be incorporated into the scan task.

```
static void appTaskScan(void *pdata) {
 uint8_t status;

 while(true) {
  OSSemPend(readyToScan, 0, &status);
  pc.printf("Scanning...\n");
  beginScanning();
  takeReadings();
  stopScanning();
  pc.printf("Stopping scan.\n");
  status = OSSemPost(readyToScan);
  OSTimeDlyHMSM(0,0,0,4);
 }
}
```

Once the readyToScan semaphore has been released, the scan task acquired it and the LIDAR begins scanning. Then the readings are stored to the buffer

so they can be written to the disk later and the LIDAR stops scanning. It then releases the semaphore so that the movement task can begin again.

11.4.3 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

In order to access the Micro SD-Card the SDFileSystem library was used, and was incorporated into the program by simplying including the main SDFileSystem header. To use the library, an SDFileSystem variable first needs to be instantiated. This variable takes five parameters, four relevant microcontroller pins and a name. Table 11.2 describes each of the four pins the SDFileSystem needs, the name of the corresponding microcontroller pin and what the pin actually is.

SDFileSystem 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. 11.2: RPLIDAR SDK files

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

```
#define MOSI  PTE3
```

```
#define MISO  PTE1
#define SCLK  PTE2
#define CS    PTE4

SDFileSystem 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 readingsBuffer 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.

```
static void writeReadings() {
 int arraySize = (sizeof(readingsBuffer)/sizeof(
    float))/2;

 // 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 < arraySize; i++) {
  fprintf(fp, "%f %f", readingsBuffer[i][0],
     readingsBuffer[i][0]);
 }

 // 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. Finally, the file is closed.

This method can now be incorporated into the robot's writing task.

```
static void appTaskWrite(void *pdata) {
```

```
 uint8_t status;

 while(true) {
  OSSemPend(readyToWrite, 0, &status);
  pc.printf("Writing readings to SD card...\n");
  writeReadings();
  pc.printf("Done writing.\n");
  status = OSSemPost(readyToWrite);
  pc.printf("Releasing write semaphore...");
  OSTimeDlyHMSM(0,0,0,5);
 }
}
```

The task pends on a readyToWrite semaphore which is released at the very end of the movement task once all movement and scanning has been performed. The write task acquired the semaphore, calls the method to write the readingsBuffer to disk and then posts the semaphore.

12. TESTING

12.1 Introduction

12.2 Initial Prototype

12.2.1 Hardware

12.2.2 Software

12.3 Implementing Requirements

12.3.1 Movement

Hardware

Software

12.3.2 Observation

Hardware

Software

12.3.3 Processing Observational Data

Hardware

Software

13. PREVIOUS FAILED ATTEMPTS

The previously described implemented methods only give a walkthrough of the implementation attempts that enjoyed the most success, as well as a brief description of any smaller issues that came up along the way. There were whole approaches to the project that ultimately failed however. The purpose of this chapter is to detail any such attempts.

13.1 LIDAR Communication

A problem encountered near the start of the project was getting the SDK to work with the LIDAR sensor. The 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 kernal header files were actually copied into the LIDAR SDK, but this just resulted in other dependency problems coming up.

Rather than continuing to spend time attempting to get the SDK working, it was decided to simply use the actual protocol that the SDK implemented. The protocol works by recieving request packets made up of certain bytes. Depending on what the bytes are, the LIDAR interprets the packet as a different command. These packets were created in the robot's program as arrays of bytes, and the Serial puts() command was used to send them to the LIDAR. getc() was then used within a task's main loop to retrieve the outputted data and it was printed to a terminal window. Relevant code snippets can be seen below.

```
Serial pc(USBTX, USBRX);
char startRequest[] = {0xA5, 0x20};
device.puts(startRequest);

while (true) {
 pc.printf(device.getc());
}
```

This had some results. Bytes were being recieved to the LIDAR but they were inconsistent, and sending a request to stop scanning would successfully stop the data output but any subsequent attempts to send the start request again would fail unless the entire program was reset. One possible fix was to implement the getc() method inside of an interrupt handler. Having getc() constantly called whenever the while loop ran might have been causing problems, but with an interrupt handler it would only trigger when there was actually something there.

```
device.attach(&rxInterrupt, Serial::RxIrq);

void Rx_interrupt() {
 pc.printf(device.getc());
}
```

This appeared to resolve the problems and values were recieved consistently. These numbers were written in a binary representation to the Micro-SD card so they could be processed by the GUI, but there were problems there as well.

This binary data would have to be turned into actual angle and distance numbers however. Fig 13.1 is from the LIDAR protocol documentation, giving a breakdown of what byte in a returned scan packet corresponds to what information.

So, for example, the first 6 bits of the first returned byte is used to store the scan's quality value. The idea behind the initial map processing software was to start at the beginning of the file containing binary scan data, then iterate through it by taking chunks of bits and storing them in variables.

```
# Read in the bits according to the LIDAR response
    structure
quality = f.read(6)
inverseStart = f.read(1)
start = f.read(1)
angle_first = f.read(7)
checkbit = f.read(1)
angle_second = f.read(8)
distance = f.read(16)
```

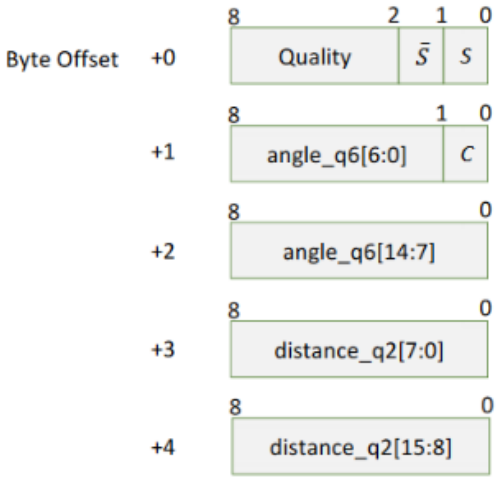

Fig. 13.1: Protocol Breakdown

One small additional tweak that needed to be made was joining the first and second angle chunks together to form the full value.

```
# Append the angle_q6 bits
angle = (b"".join([angle_first, angle_second]))
```

The protocol documentation states that these aren't the true values however. The actual angle value is the binary value divided by 64 degrees, and the actual distance value is the binary value divided by 4 millimeters.

```
# Turn binary into decimal
# 'Actual heading = angle_q6/64.0 Degree'
angle = (int(angle, 2) / 64.0)
# 'Actual Distance = distance_q2/4.0 mm'
distance = (int(distance, 2) / 4.0)/100
```

These points were then turned into mappable X and Y values. Despite all of this, none of this worked as intended. The produced maps made no logical sense, here is a map produced via this method from scans obtained from the inside of a rectangular box.

The generated values were printed, and it was observed that every now and then angles would be impossible values above 360. After all of this it was decided to go back to the drawing board and attempt to get the SDK working again since using the protocol was rapidly leading nowhere and burning time. Searching around on the mbed website stumbled across a robotic program that made use of a similar LIDAR system by SLAMTEC. The SDK in use there was a modified version that seemed to work fine with embedded systems, but still had all the necessary open licensing. This SDK

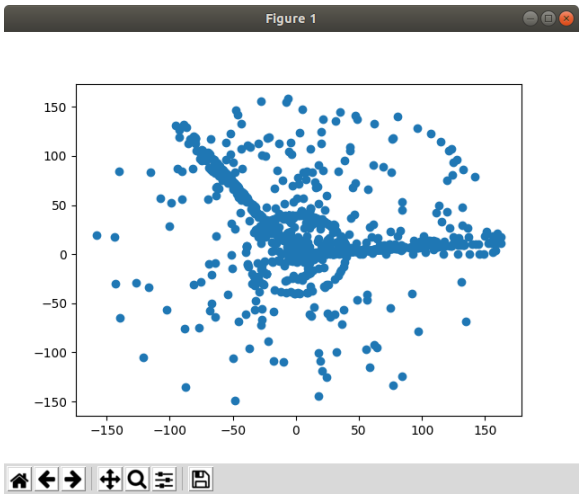

Fig. 13.2: Resulting map of a box

was tested with the program and compiled fine, and is what was used to ultimately implement the robot's observational capability.

Part III

EVALUATION

14. INTRODUCTION

15. PRODUCT EVALUATION

16. PROCESS EVALUATION

17. CONCLUSIONS AND RECOMMENDATIONS

Part IV

APPENDICES

BIBLIOGRAPHY

[1] *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).

[2] 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.

[3] 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` (Accessed 12 December 2018).

[4] Andrea Censi. An icp variant using a point-to-line metric. 2008.

[5] 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.

[6] 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).

[7] 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.

[8] 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).

[9] Owen Edwards. URL: https://os.mbed.com/users/Owen/code/nRF24L01P/ (Accessed 11 January 2019).

[10] Jean J Labrosse. *MicroC/OS-II: The Real Time Kernel*. CRC Press, 2002.

[11] mbed official. URL: https://os.mbed.com/users/mbed_official/code/SDFileSystem/ (Accessed 11 January 2019).

[12] *RPLIDAR A2 - Introduction and Datasheet*. SLAMTEC. Available at: https://www.robotshop.com/media/files/pdf2/ld208_slamtec_rplidar_datasheet_a2m8_v1.1_en_2_.pdf (Accessed 14 December 2018).

[13] SLAMTEC, . URL: http://bucket.download.slamtec.com/4ad536379886737a0ca660d4cf650a896a7ea09f/LR001_SLAMTEC_rplidar_protocol_v2.1_en.pdf (Accessed 10 January 2019).

[14] SLAMTEC, . URL: https://www.slamtec.com/en/Lidar/A1 (Accessed: 06 December 2018).

[15] Cyrill Stachniss. Introduction to robot mapping, October 2013.

[16] Sebastian Thrun and John J Leonard. Simultaneous localization and mapping. *Springer handbook of robotics*, pages 871–889, 2008.

[17] 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.

[18] 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.

[19] 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.