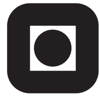


NTNU – Trondheim
Norwegian University of
Science and Technology

ITS using LEGO Mindstorm

Kristian M Overskeid

Master of Science in Communication Technology
Submission date: March 2015
Supervisor: Frank Alexander Krämer, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem description

Intelligent transportation systems (ITS) are advanced services and infrastructures enabling safer, more efficient, smarter and more environment friendly use of transport networks. The most public known ITS technologies today help various users to make better decisions in traffic by increasing the access to relevant information.

This project will focus on Personal Rapid Transit (PRT) systems. The system consists of small automated vehicles, called podcars, operating on a network of specially built guide ways. LEGO Mindstorms EV3, LEGO city train, Raspberry Pi and NFC readers will be used to build a lab to visualize services and infrastructures. The use of LEGO makes it easy to physically change the design of the lab. This requires a software design capable of handling changes in the physical layout of the tracks and the number of podcars. To make the lab usefull for further development and testing, the software should be designed in an intuitive way making it easy to change the system design. To accomplish this, Reactive Blocks will be used when programming.

To make the lab relevant, the design solutions should be based on real world considerations as long as it's possible to implement with LEGO. To ensure this, the system should be based on planned or already implemented PRT systems. The system design of the pods should aim at making their behaviour as natural as possible.

The project will only focus on ICT part of the PRT system. This includes the infrastructure in which the ICT is dependent on.

The objectives of this project are:

- Build a LEGO rail track with automatic switches and LEGO trains (podcars) capable of regulating the speed, detect obstacles in the way and be aware of the position.
- Design a system software able to route the podcars in an effective and safe way between stations. There should be a simple and intuitive way to adapt the software to changes in the physical layout of the track. Changes to the number of podcars should be handled seamlessly. The behaviour of the pods should be as realistic as possible.
- Design a program able to simulate customers ordering the podcars to different stations.

– If there is time, test and analyze different system designs and track layouts.

Abstract

Intelligent Transport Systems (ITS) is a general term for information-and commuication systems aimed at making the transport sector more safe, efficient and environmentally friendly. During the recent years, cars have been equipped with numerous ITS applications such as parking assistant, cruise control and even collision avoidance systems directly interfering with the car to avoid an imminent collision. Car companies claim that autonomous cars, i.e. cars capable of driving without human interference, will be on the market by the end of the decade.

Private Rapid Transport (PRT) is an emerging public transportation branch combining several different applications developed for ITS. The main difference between PRT and more tradition public transport systems is that the vehicles are operating on demand. When customers places an order, the system will summon the closest vehicle, which will pick up the customers and deliver them at the desired destination automatically. To avoid interference with other vehicles and pedestrians, the vehicles will run along specially built guideways usually separated by levitation. The goal is to make PRT a realistic alternative to the private car by offering close to the same privacy and flexibility. To offer the same flexibility as a car, the system must provide sufficient departure and destination stations. To achieve this, the guideways are intended to form a mesh network scattered all over cities and their suburbs. However, no operational PRT systems with more than five stations exist today.

Lego offers a wide range of products spanning from more or less realistic models of trains and other vehicles to advanced programmable robots. Mindstorms is Legos robotic theme, which was introduced in 1998. In this thesis, I have used EV3, which is the third generation of Lego Mindstorms. EV3 features many different advanced sensors and both powerful and accurate servo motors that can be connected to and controlled by the EV3 intelligent brick. By installing the Lego Java Operating System (LeJOS) on the intelligent brick, it is possible make programs in Java and use all the libraries included in the Java 7 Standard Edition. In addition, LeJOS makes the intelligent brick support WiFi communication, leaving developers with their imagination as the biggest restriction.

The different pieces are designed to be compatible with each other in some way, regardless of age or theme. This means that a simple Lego toy intended for children under the age of ten can be rebuilt and combined

with Lego Mindstorms to become a sophisticated robot. In this thesis I have rebuilt three carriage Lego City trains into a single carriage PRT vehicles, referred to as pods.

The purpose of this thesis is to determine whether or not it's possible to simulate a real world PRT control system using Lego Mindstorms in combination with Lego City trains. Because it seems like no one else have ever attempted to do this, I have made a functional PRT system on my own. The process is thoroughly described in this report and the work is mainly divided into two objectives:

1. Design a pod that can drive to a given location with smooth and realistic movements
2. Design a higher level control system that can command the pods to move efficiently and safely between stations.

1. Although I used the physical pod design from my previous project at NTNU, designing the pods control program to keep track of the pods exact location was the most time consuming individual challenge in this project. Early on, I decided to use the color sensor to count sleepers, where each sleeper in the guideway represents a unique location. Because I could not find any other projects using the Lego color sensor for a similar purpose, I had to interpret the raw data from the color sensor and process it into reliable location data on my own. When the color sensor was able to detect each sleeper exactly once, it was relatively simple to calculate the speed of the pod because I knew both the distance and time between each sleeper. To make the pod stop smoothly at a given location, I decided to use a quadratic function to calculate the required braking distance based on the current speed. I also tested different approaches, but without the same consistency as the quadratic function. In the end, the result is a pod stopping at a given location with an accuracy of 10 cm. It accelerates and brakes smoothly. To communicate with a higher level control system, I have chosen to use MQTT.

2. I have chosen to use a control system architecture consisting of a *central*, multiple *guideway controllers* and one switch operator per guideway controller.

The *guideway controllers* are responsible of one control area each. There is one control area per switching intersection in the guideway and pods can only enter a control area when ordered to by the guideway controller in charge of that control area. Because the guideway controllers

are the only units that can allow a pod to move, it is responsible for the safety, i.e. make sure the pods do not crash into each other. In addition, the guideway controller sends switch commands to the switch operator located in its area. When a pod approaches a switching intersection, the guideway controller looks up the pods destination station in a table and orders the switch operator to switch the tracks in the correct location to route the pod the shortest path.

The *switch operator* receives switch orders from the guideway controller. The switch operator is always aware of the tracks direction. If the tracks are in the correct direction when it receives a switching order, no further action is taken. If not, the tracks are switched in the correct direction by rotating a motor connected to the switch handle.

The *central* was intended to assign orders to pods automatically and constantly monitor the position of each pod to be able to detect congestion and update the routing tables. Unfortunately, I did not have time to make the central automatic. However, I have made a simple graphical user interface where an operator manually can assign orders to pods. An order can be set to repeat, which means that the pod will drive constantly between the two given stations. By assigning multiple pods repeated orders to move between different stations, a lot of interesting situations occurs at both station tracks and merging intersections.

When I tested the control system by only using a simple computer simulation, I was able to reveal some logical errors. When I had corrected all these errors, I tested the control system in the lab. This instantly exposed a lot of situations where the system failed, causing deadlocks where one or more pods stopped at wrong locations because it wasn't ordered to continue by the control system. After analyzing the results, I was able to correct most of the errors. However, I did not have time to make the system work perfectly. Nevertheless, most of the times, the system now works as intended for many minutes before an error situation occurs. By using the lab, I was able to expose errors I couldn't detect using computer simulations. In addition, it was much easier to understand the source of the errors from observing the pods behaviour. Based on this experience, I have found that Lego Mindstorms is sufficiently advanced and adaptable to simulate a simple PRT control system.

Samandrag

Intelligent Transport System (ITS) er ei felles nemining som omfattar system som tek i bruk informasjon- og kommunikasjonsteknologi for å gjera transportsektoren meir trygg, effektiv og miljøvennleg. I løpet av dei siste åra har bilar blitt utrusta med mange ITS funksjonar, som til dømes automatisk parkerings assistent, cruice kontroll og endåtil kolisjonsavvergingssystem som grip inn og tek kontroll over bilen for å unngå kollisjonar. Bilprodusentar meiner at autonome bilar, dvs bilar som er kapable til å køyra utan sjåfør, vil vera på marknaden innan utgongen av tiåret.

Private Rapid Transport(PRT) er eit offentleg transportsystem i rask utvikling som kombinerer mange ulike funkjsonar som er utvikla for ITS. Hovudforskjellen mellom PRT og meir tradisjonelle offentlege trasportsystem er at kjøretøya opererer på førespurnad. Når ein kunde bestiller trasport, vil systemet senda det næraste kjøretøyet for å plukka opp og kjøra kunden til den ønska stasjonen atuomatisk. For å unngå interferens med andre kjøretøy og fotgjengerar, vil kjøretøya nytta eigne spesialbygde køyrebanar som vanlegvis er heva over bakkenivå. Målet er å gjera PRT eit realistisk alternativ til privat bil ved å tilby tilnærma like mykje privatliv og fleksibilitet. For å tilby den same fleksibiliteten som ein bil, må systemet tilby tilstrekkeleg mange stasjonar. For å oppnå dette, er hensikten å oppretta eit maskenettverk spreidd over byar og tilhøyarde forstadar. I dag fins det ingen PRT system med med meir enn fem stasjonar som er sett i drift.

Lego tilbyr eit vidt spekter av produkt som omfattar alt frå meir eller mindre realistiske modellar av tog og andre kjøretøy, så vel som programmerbare robotar. Minstorms er Lego sitt robot-tema, som var introdusert i 1998. I denne oppgåva har eg nytta EV3, som er den tredje generasjonen av Lego Mindstorms. EV3 omfattar mange ulike avanserte sensorar og både krftige og nøyaktige servo-motorar som kan koplast til og kontrollerast av den intelligente brikka tilhøyrande EV3. Ved å installera Lego Java Operating System (LeJOS) på den intelligente brikka, er det mogleg å laga Java program og nytta alle biblioteka inkludert i Java 7 Standard Edition. I tillegg vil LeJOS gjera at den intelligente brikka støttar kommunikasjon via WiFi, noko som levnar lite anna enn fantasien som hindring.

Dei ulike Lego delane er utforma for å passa saman med kvarandre på ein eller annan måtte, uansett alder eller tema. Dette betyr at eit

simpelt Lego leketøy meint for ungar under ti år kan bli bygd om og kombinert med Lego Minstorms til ein avansert robot. I denne oppgåva har eg bygd om Lego sitt trevogners tog til eit enkeltvogs PRT kjøretøy som blir referert til som pod.

Formålet med denne oppgåva er å sjekka om det er mogleg å simulera eit ekte PRT-kontrollsystem ved å nytta Lego Mindstorms i kombinasjon med Lego City tog. På grunn av at det kan sjå ut som at ingen andre nokon sinne har prøvd å gjera dette, har eg laga eit fungerande PRT system på eigenhand. Prosessen i nøye skildra i denne rapporten og arbeidet er hovudsakleg delt opp i to mål:

1. Designa ein pod som kan kjøra til ein gitt lokasjon med rolege rørsler
2. Designa eit kontrollsystem som kan kommandera podane til å kjøra trygt og effektivt mellom stasjonane

1. Sjølv om eg gjenbrukte det fysiske designet av poden frå prosjektet mitt ved NTNU, var det designet av eit kontrollsystemet med moglegheit til å halda styr på poden sin loaksjon som var den mest tidkrevjande og utfordrande enkeltoppgåva. Eg bestemte meg tidleg for å nytta fargesensoren til å tella sviller, der kvar svilla i kjørebanen representerer ein unik lokasjon. På grunn av at eg ikkje fann nokon andre prosjekt som nyttar Lego sin fargesensor på tilvareande måte, måtte eg tolka rådata frå fargesensoren og gjera dei om til pålitlege lokasjonsdata på eigenhand. Når fargesensoren kunne detektera kvar sville nøyaktig ein gong, var det relativt enkelt å rekna ut farten til poden fordi eg visste både avstand og fart mellom kvar svilla. For å få poden til å stoppa roleg ved den gitte lokasjonen, bestemte eg meg for å nytta ein kvardratisk funkjson for å rekna ut den naudsynte stopplengda ut frå den målte hastigheita. Eg testa også andre moglege framgongsmåtar, men desse viste seg å ikkje fungera like bra. Til slutt endte eg opp med eit sluttresultat der podane stoppar ved den gitte lokasjonen med eit avvik på 10 cm. Poden aksellererer og bremsar med jevne rørsler. For å kommunisera med kontrollsystemet har eg valgt å nytta MQTT.

2. Eg har valgt å nytta ein kontrollsystemarkitektur beståande av ein *sentral*, fleire *kjørebanekontroller* og ein *svitsjeoperatør* for kvar *guideway controller*.

Kjørebanekontrollarane er ansvarlege for eit kontrollområde kvar. Det er eitt kontrollområde for kvar svitsjekryss i kjørebanen og podar kan kun kjøra inn i eit kontrollområde ved ordre frå kjørebanekontrolleren

ansvarlig for kontrollområdet. Fordi kjørebanekontrollarane er dei einaste einingane som kan gi podar lov til å kjøra, er dei ansvarlege for tryggleiken, mao sørga for at podane ikkje kan krasja inn i kvarandre. I tillegg sender kjørebanekontrollarane kommandoar til svitjseoperatørane tilhøyrande området deira. Når ein pod nærmar seg eit svitsjekryss, slår kjørebanekontrolleren opp stasjonen poden skal til i ein rutingtabell og gir svitsjeoperatøren ordre om kva for ein posisjon spora skal vera i for at poden blir ruta den kortaste vegen.

Svitsjeoperatøren mottek svitsjeordrar frå kjørebanekontrolleren. Svitsjeoperatøren er klar over kva for ein posisjon spora er i. Dersom spora er i korrekt posisjon når ein svitsjeordre blir motteke, vil ingenting bli endra. I motsett tilfelle, vil spora bli endra til den korrekte posisjonen ved å rotera ein motor som er kopla til sporpensen.

Intensjonen var at *sentralen* skulle tildela ordrer automatisk til podane og konstant overvaka posisjonen til kvar enkelt pod for å kunna oppdaga område med trafikkork og oppdatera rutingtabellane. Eg hadde dessverre ikkje tid til å laga ein automatisk sentral. Eg laga likevel eit simpelt brukargrensesnitt der ein manuelt kan tildela ordre til podane. Ein ordre kan bli sett til å bli sent om att uendeleg mange gonger slik at poden køyrer konstant mellom to stasjonar. Ved å tildela flerie podar slike ordre, oppstår det mange interessante situasjonar både ved stasjonar og kryss.

Då eg testa kontrollsystemet ved å nytta ein simpel datasimulasjon, oppdaga eg logiske feil. Etter å ha retta desse feila, testa eg kontrollsystemet i laben. Dette avslørte umiddelbart mange situasjonar der systemet feila, noko som førte til at podane ikkje fekk tildelt nye ordre og vart ståande i ro. Etter å ha gått gjennom resultata, klarte eg å retta flesteparten av feila. Eg hadde likevel ikkje tid til å få systemet til å fungera feilfritt. Til trass for dette fungerer systemet stort sett i fleire minutt før det oppstår feilsituasjonar. Det å nytta laben oppdaga eg fleire feil som eg ikkje fann i datasimulasjonen. I tillegg var det mykje lettare å forstå feilkjelda ved å observera oppførselen til podane. Basert på denne erfaringa, har eg funne ut at Lego Mindstorms er tilstrekkeleg avansert og fleksibelt til å simulera eit PRT-kontrollsystem.

Preface

This master thesis concludes my master's degree in Communication Technology with emphasis in system development at the Norwegian University of Science and Technology (NTNU), department ITEM. The thesis was carried out during winter of 2014/2015.

Trondheim, 2015-03-16

Kristian Myrland Overskeid

Contents

List of Figures xvii

List of Tables xxi

List of Algorithms xxiii

1 Introduction 1

1.1 Intelligent Transport Systems and Personal Rapid Transport 1

1.1.1 Challenges of Personal Rapid Transport Systems 2

1.1.2 Simulation for PRT 2

1.2 Problem Description and Scope 3

1.3 Structure of the Report 4

2 Background 7

2.1 Intelligent Transport Systems 7

2.2 Private Rapid Transport 7

2.3 Lego Mindstorms 9

2.4 Communication Protocols 10

2.4.1 Transmission Control Protocol 11

2.4.2 Message Queue Telemetry Transport 12

2.5 Lego Java Operating System 13

2.6 Reactive Blocks 14

2.7 Project at NTNU 14

3 Initial Analysis 17

3.1 Choosing ITS Application 17

3.2 System Requirements 17

3.2.1 Functional Requirements 17

3.2.2 Non-functional Requirements 18

3.3 Using Reactive Blocks and MQTT 18

3.4 Choosing Distributed and Asynchronous Control 18

4 Physical Components 21

4.1 Building a Guideway From Lego Train Tracks 21
4.1.1 Guideway Layout 21
4.1.2 Automatic Switching of the Pods 23
4.2 Pod Components 23
4.2.1 Color Sensor 24
4.2.2 Propulsion Control 25

5 Testing the First Version of the Pod Control System 27
5.1 Initial System Design 27
5.2 Only Detecting Blue Segment Dividers 27
5.3 Detecting the Same Segment Divider Multiple Times 28
5.4 Too Long Stopping Distance 28
5.5 Braking distance 29
5.6 Reaction Distance 29
5.7 Reaction Distance vs Braking Distance 31
5.8 Reducing the Rotation Time 32

6 Speed and Position Testing 33
6.1 Utilizing the Potential of the Color Sensor 33
6.2 Position Measurement 34
6.2.1 Description of the Color Sensor Thread 34
6.2.2 Testing the Color Sensor 34
6.2.3 Checking the Time Difference Since the Last Color Change . 35
6.2.4 Require Consecutive Color Samples 35
6.2.5 Adding LastRegisteredColorID to the Origial Code 36
6.2.6 Intersections and Flexible Tracks 38
6.3 Speed Measurement 39
6.3.1 Speed Calculation 39
6.3.2 Selecting the Value of Δs 40

7 Stopping and Starting Procedure 47
7.1 Stopping procedure 47
7.1.1 Calculate the Stopping Distance by Standard Formula 47
7.1.2 Lookup the Stopping Distance in a Table 48
7.1.3 Calculate the Stopping Distance by Quadratic Function . . . 48
7.2 Avoid Pre-intended Stopping 49
7.2.1 Detect Stop 50
7.3 Starting Procedure 51

8 Abstract Guideway Design 53
8.1 Accurate Position Awareness 53
8.2 Dividing the Guideway in Number Series 53

8.2.1 Sleeper Number Assigment in Intersections 54
8.2.2 Adjustment Sleepers 54
8.3 Correction Sleepers 55
8.4 Initial Parameters 56

9 Control System Design 59
9.1 Using Embedded System Design 59
9.1.1 Distribution of Responsibility 59
9.1.2 Essential Interactions 61
9.2 Using Distributed Control 62
9.2.1 Dividing the Guideway into Control Areas 62
9.2.2 Handover Procedure Between Control Areas 64

10 Interactions 65
10.1 Operational Communication 65
10.1.1 Movement Authority 65
10.1.2 Switch Orders and Confirmations 65
10.1.3 Location Update 66
10.1.4 Arrival Notice 66
10.1.5 Handover 66
10.1.6 Destination Request and Responce 67
10.1.7 Adjustment Sleeper Information and Confirmation 68
10.2 Administrative Communication 68
10.2.1 New Order, Cancel Order and Order Status 68
10.2.2 Route Updates 70
10.3 Initial Communication 70
10.3.1 Registration, Control Area, Station Name and Pod Id 71
10.3.2 Movement Authority and Handover 71

11 The Guideway Controllers Internal Functionality 73
11.1 MQTT Topic Hierarchy 73
11.2 Dividing the Control Area Into Track Sections 74
11.3 Reservations 75
11.4 The Guideway Controller Block 76
11.4.1 MQTT Subscribe 76
11.4.2 MQTT Publish 77
11.5 The Control Unit Block 78
11.5.1 Classify Message 78
11.5.2 Format Message 80
11.5.3 Route Manager 81
11.5.4 Switch Manager 81
11.6 The Guideway Manager Block 82

11.6.1 Update Location(1) 82
11.6.2 Select Action(2) 84
11.6.3 The Yellow and Black Flow 85
11.6.4 Registering an Approaching Pod(green flow) 86
11.6.5 Switch an Approaching Pod(dark blue flow) 86
11.6.6 Merge an Approaching Pod(light blue flow) 87
11.6.7 Check Available Sleepers(brown flow) 87
11.6.8 Move Pod(purple flow) 87
11.6.9 Unregister Pod(red flow) 88
11.7 The Movement Authority Manager Block 88
11.7.1 The Loop(yellow flow) 89
11.7.2 Generating and Sending a New Movement Authority(green flow) 90
11.7.3 Generating and Sending a Handover(blue flow) 90
11.7.4 GetStartSection 91
11.7.5 Reserve Sleepers 91

12 The Pods Internal Functionality **95**
12.1 The Pod Block 95
12.1.1 The Start-up Procedure(green flow) 95
12.1.2 Propulsion Controller Block(2) 96
12.2 Pod Controller Block 96
12.2.1 The Generate Message Block 96
12.2.2 The Order Manager Block 97
12.2.3 The Generate Message Block 98
12.2.4 The Filter Reports Block 98
12.3 The Movement Authority Manager Block 98
12.3.1 Passing a New Sleeper(yellow flow) 99
12.3.2 Stopping the Pod(red flow) 99
12.3.3 Stopping the Program(dark blue flow) 100
12.3.4 New Movement Authority(green flow) 100
12.3.5 Reducing the Speed(purple flow) 100
12.3.6 Increasing the Speed(light blue flow) 100
12.3.7 Standby(black flow) 101
12.3.8 The Brake Block(1) 101
12.3.9 The Timeout Block(2) 102
12.4 The Location Manager Block 102

13 Testing and Evaluating the System **105**
13.1 Limitations 105
13.1.1 Reduced Guideway Size 105
13.1.2 Manual Central 106
13.2 Test Setup 107

13.3 Evaluating the Performance 107
13.3.1 The Pod Reports an Incorrect Location 107
13.3.2 The Pod Misses Important Messages 108
13.3.3 Expected Behaviour at Normal Operation 109
13.4 Using a Loosely Coupled System Design 110

14 Conclusion and Recommendations for Future Work 113
14.1 Conclusion 113
14.1.1 Develop Lego Pods With Realistic Behaviour 113
14.1.2 Develop a System to Control the Pods In a Safe and Efficient Manner 115
14.2 Recommendations for Future Work 116

References 117

List of Figures

2.1 ITS applications and their relation to safety, efficiency and environment 8
2.2 Pictures of the Vectus and ULTra PRT systems 10
2.3 The LEGO Mindstorms EV3 Storm Robot 11
2.4 Protocol layering 12
2.5 The MQTT publish/subsrcibe procedure 13
2.6 An example of a Reactive blocks building block 14
2.7 Reactive Blocks reveals an ESM violation 15

4.1 The final guideway layout 22
4.2 Station and transit track 22
4.3 Different intersections 23
4.4 Picture of the switch operator 23
4.5 Picture of the pod from different angles 24

5.1 Reaction distance vs braking distance. The red vertical line represents the distance covered when the motor is set to speed level -2. 31
5.2 LEGO gear 32

6.1 Test track. There are 32 sleepers between each red sleeper 33
6.2 Small test track 35
6.3 Small test track with different color between the sleepers 38
6.4 Picture of two intersections 38
6.5 Picture of a flexible track between two straight tracks 39
6.6 Different values of Δs 40
6.7 Speed with Δs set to alternative B. The pod travels two rounds around a test track. The different rounds are represented by different colors. Ideally, the two graphs should be identical. 41
6.8 Speed with Δs set to alternative C. The pod travels two rounds around a test track. The different rounds are represented by different colors. Ideally, the two graphs should be identical. 42

6.9 Speed with Δs set to alternative D. The pod travels two rounds around a test track. The different rounds are represented by different colors. Ideally, the two graphs should be identical. 42
6.10 Acceleration with all the different Δs alternatives. The black line represents alternative B, red alternative C and blue alternative D. 43
6.11 Retardation with all the different Δs alternatives. The black line represents alternative B, red alternative C and blue alternative D. 44
6.12 Measured speed at speed level 2 to 5 45

7.1 Test track for measuring stopping distances 49
7.2 Stopping distance at from different speeds 49

8.1 Example of how the number series can be organized in intersections . . 54
8.2 Illustration of the guideway divided in number series 56
8.3 Illustration of how the number series in intersections are divided when using the maximum number of number series 57

9.1 Illustration of the control systems differnt layers of responsibility 60
9.2 The most essential interactions between the different units 61
9.3 The guideway split into control areas 63
9.4 Illustration of how the control areas are divided in intersections 63
9.5 Illustration of the handover proceudre 64

10.1 Illustration of the communication between a pod and a guideway controller 66
10.2 Illustration of the communication between two pods and a guideway controller 67
10.3 Sequence diagram of the handover procedure 69
10.4 Order life cycle 70

11.1 The division of sections at switching intersections 74
11.2 The division of sections at merging intersections 75
11.3 Illustration of the reservation priority between pods 76
11.4 Screen dump of the guideway controller block 77
11.5 Screen dump of the control unit block 78
11.6 Screen dump of the classify message block 78
11.7 Message buffering and filtering activity diagram 79
11.8 Message formatting activity diagram 80
11.9 Adding and removing pods activity diagram 80
11.10 State Machine Diagram of the switch manager. 81
11.11 Screen dump of the guideway manager block 82
11.12 Screen dump of the update location block 83
11.13 Screen dump of the select action block 84
11.14 Screen dump of the movement authority manager block 89

11.15 Screen dump of the get first section block 92
11.16 Screen dump of the reserve sleepers block 93
11.17 Step by step illustration of the reservation process 93

12.1 Screen dump of the pods system block 95
12.2 Screen dump of the pod controller block 97
12.3 Screen dump of the movement authority manager block 99
12.4 The state machine diagram of the ”brake” block 102
12.5 The location manager block. 103

13.1 The reduced guideway design used in the lab 106

List of Tables

7.1 The time period the color sensor goes to sleep every time it detects a color change 51

List of Algorithms

6.1 Original code 35
6.2 Checking the time difference since the last color change 36
6.3 Require consecutive color samples 37

Chapter 1 Introduction

1.1 Intelligent Transport Systems and Personal Rapid Transport

During the last decades, Intelligent Transport Systems have gradually been implemented in vehicles and roadside infrastructure to increase the safety and efficiency to road users as well as automating traffic and mobility management. Even though ITS generally is associated with road transport, other modes of transport apply ITS technologies in various extent. One of these modes are personal rapid transport systems(PRT), which futures many ITS technologies.

Although PRT is an emerging public transportation branch with a lot of new and exciting solutions, the concept originates from the 1950s. This decade is characterized as pivotal for the car industry, and as suburbs in the US developed further away from city centers, the cities improved the roads and highways. As a result, personal car ownership rapidly replaced public transport, causing significant air pollution problems. To address the problem, PRT was pointed out as the only public transport systems offering the same flexibility and private sphere as a car, thus the most likely public transportation mode to be chosen if favour of the car.

Even though PRT has existed as a theoretical concept for more than half a century, there are no existing examples of real world PRT systems with a wide variety of destination stations, offering sufficient flexibility to be an actual alternative to the car. Certainly, a few PRT systems are operational, but even if these systems offer on demand service and private sphere, they have too few stations to provide more flexibility than a traditional public transport covering 2-3 stations at a frequent interval.

1.1.1 Challenges of Personal Rapid Transport Systems

While the first PRT projects were limited by insufficient technology solutions, both in the area of information and communication technology (ICT) as well as construction materials, the cause of failure in more recent PRT projects is more vague. Studies suggest that modern ICT solutions and material technology can lower the investment and operational costs to less than half of existing public transport systems. The vision of PRT is to offer a wide range of stations connected by a mesh network of guideways in both downtown and rural areas, offering the same flexibility as a private car.

Considering the technological and theoretical financial feasibility compared to competing branches of public transport, why are there currently so few examples, if any, of operational PRT systems? In addition, the environmental benefits compared to traditional transportation modes should make PRT and obvious choice. Supporters blame lack of vision in public administrations, partially due to influence from lobbyists representing competing traditional public transport industries as well as political reluctance to non-proven technology. Opponents, on the other hand, refers to the history of failed PRT projects and argues that the extreme light weight and flexibility of PRT constraints the passenger capacity to a level where the investment and operational costs cannot be justified even with a functional control system. Regardless of the underlying causes, the funds to many PRT projects has been cancelled prior to the completion of a functional test system, effectively shutting the projects down.

1.1.2 Simulation for PRT

A highly adaptable and cheap miniature prototype of a PRT system would offer an intermediate stage between computer simulations and costly test installation. This could potentially help developers to detect and improve errors at an early stage. Such a model would also contribute to giving investors and decision makers a more definite, unambiguous and perhaps convincing impression of a PRT system at an insignificant price.

Making an exact small scale model of a PRT system is an ambitious and time consuming task because there are no off-the-shelf components. This means that the model of the guideway and pods needs to be designed and made from scratch or customized from similar existing products, such as model trains. In both cases, building an exact down-scaled model would require a lot of resources. In addition, at an early stage, the physical design is usually subject to frequent alterations, which would also involve changes to the customized model components.

1.2 Problem Description and Scope

Lego offers a wide range of products providing everything from realistic models of vehicles to advanced programmable robots. The different pieces, regardless of age or theme, are all compatible with all other existing pieces in some way. The design of Lego pieces makes them easy to assemble and connect in different ways, and at the same time just as easy to dismantle.

Most people agree that Lego is an amusing toy, but would Lego also be suitable to model and test an early stage PRT system? Although a lot of different advanced Lego projects using Lego Mindstorms can be found by a quick google search, none of them seems to be made as a test model for a real world system. It is also hard to find documentation of Lego projects controlling multiple units in a distributed manner, which is one of the fundamental principles in the system I will present in this report.

The best way to figure out whether Lego can be used to build a relevant model of a PRT system is to actually try to build one. This is the main focus of this report. While Lego is cheap, versatile and easy to modify, it does not offer the possibility to build a scaled model of pods or guideways. This means that different mechanical designs, such as propulsion method, braking systems and switching mechanism cannot be tested with Lego. On the other hand, the wide range of sensors compatible with Lego Mindstorms and the fact that the latest model, EV3, can communicate via WiFi, should make it applicable for testing the Information, Communication and Control System (ICCS) of a PRT system.

The purpose of this thesis is to determine if it's possible to build a Lego PRT platform and use it in the context of testing and evaluating the ICCS of a real world PRT system. To do this, I have rebuilt the three carriage Lego City trains into single carriages to simulate pods. The pods move along Lego City tracks, which consists of intersection and turn segments, making it possible to build a copy of any real world guideway layout. The pods contains a programmable EV3 brick capable of communicating with any WiFi enabled unit. The pods are aware of their exact location and speed via a sensor and is able to control the speed. Motors connected to the track intersections otherwise manual switch handles permits automatic switching of pods.

The first objective is to make a pods move to a specific and accurate position while reporting it's position continuously as it moves along the tracks. The behaviour must be smooth, i.e. it should be perceived as comfortable to be a passenger on board the pod. I have solved this by making the pods "stupid" in the sense that they are not aware of the position of stations, intersections or other pods. They will simply move a certain distance from their current position at a given speed when commanded to by a higher level control system. The pods propulsion control makes

sure that the pods accelerate and decelerate smoothly and stops exactly at a given position.

The second objective is to make sure the pods can be individually controlled by a higher level control system. To check that this is actually possible, I have designed a simple control system that will guide the pods the shortest route to their destination without crashing. A simple graphical user interface will allow an operator to simulate customers ordering transport between stations.

I have chosen to use a system architecture consisting of a central that will initialize the pods and assign orders to them continuously. The guideway consisting of Lego tracks is logically divided into control areas supervised by one guideway controller per control area. Each guideway controller will be in charge of one switch operator, which is responsible of switching the tracks direction at switching intersections. No pods will be allowed to move into a control area without an approval from the guideway controller in charge of that control area. This solution does not increase the load on each guideway controller if the guideway is expanded, making it highly scalable.

While designing the pods control program, I spent a lot of time and effort to get accurate speed and position measurements from the color sensor. Because I couldn't find any other projects using the color sensor for this purpose, I had to interpret the raw data from the color sensor and process them into accurate speed and position measurements. In the start, this implied an experimental approach where I soon discovered that the high samle rate of the color sensor in combination with relatively large area included in each sample, made the returned color higly unreliable in the transition between different colors. To deal with this, I had to make the color sensor ignore false color samples, which proved to be harder than I initially assumed.

When I tested the control system in the lab, it revealed a lot of logical errors I couldn't find during computer simulation. This is interesting because it indicates that a lab simulation consisting of physical components are likely to expose errors that aren't detected by a computer simulation.

1.3 Structure of the Report

Chapter 2 provides the background information that might be useful when studying the work done in the next chapters.

Chapter 3 describes the decisions and analysis done before the process of designing the system was started. The choices are substantiated by a brief argument.

Chapter 4 explains the physical components included in the system, which includes the pods, guideway and switch operators.

Chapter 5 briefly explains the testing of the first version of the pod control system and the three main problems that occurred. Each problem is analysed before a solution or possible solution is described.

Chapter 6 describes the process of testing the color sensors ability to keep track of the pods position and measure it's speed.

Chapter 7 explains how the accurate speed and position measurements can be used to improve the system design. The main focus of this chapter is to discuss and test which approach will stop the pod in the most accurate and smooth way,which is done in the first section.

Chapter 8 explains how the system keeps track of the pods, how the guideway is divided logically to not mix the pods positions and the initial parameters needed to be entered manually into the systems when the guideway design has been altered.

Chapter 9 provides a general understanding of the control system. The different units, their responsibilities and the most essential interactions are explained. This chapter also illustrates and explains how the guideway is divided into different control areas.

Chapter 10 lists all the interactions, i.e. the communication protocols, between the different units.

Chapter 11 thoroughly explains the internal functionality of the guideway controller program.

Chapter 12 thoroughly explains the internal functionality of the pods control program.

Chapter 13 explains the lab setup and how the system performed.

Chapter 14 summarizes the process of making designing and implementing the system, the challenges I encountered, as well as the systems performance in the lab.

Chapter 2
Background

This chapter offers background information that might be useful when studying the work done in the next chapters.

2.1 Intelligent Transport Systems

Intelligent Transport Systems is a wide range of applications aimed at making the transport sector more safe, efficient and environmental friendly. While some ITS applications can be categorized as futuristic and only exist on paper, others are used extensively every day. While the general public is aware of the most known ITS applications, such as GPS navigation, electronic toll collection and speed cameras, other ITS applications not interacting directly with humans, are not that commonly known. Such applications might include traffic light optimization, road condition monitoring and intelligent cargo.

2.2 Private Rapid Transport

PRT is an emerging public transportation branch which combines several different ITS applications. A PRT system consists of small autonomous vehicles, called pods, running along specially built guideways. Each vehicle has room for 2-6 passengers, which will give the passengers a more personal sphere than public mass transport.

While most traditional public transport systems operate at fixed schedules, PRT operates on-demand. This means that the pods stays idle until they are summoned by a customer, which has a fortunate effect on off-peak capacity utilization unlike traditional public transport systems.

To avoid potentially dangerous situations and increase efficiency, the guideway is separated from other traffic and pedestrians. This is usually solved by building the guideway some floors above ground level. The separation allows the pods to move at speeds between 40-70 km/h, which is relatively fast considered that the pods can

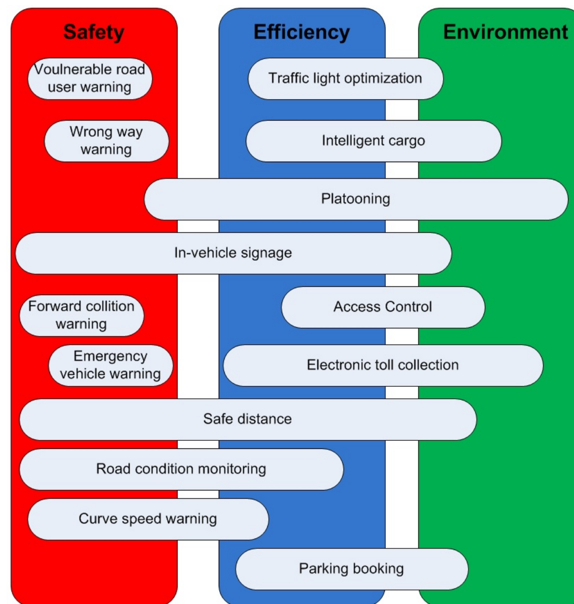


Figure 2.1: ITS applications and their relation to safety, efficiency and environment. The figure is from my project report [1]

keep this speed throughout the whole journey. Because the pods are relatively small, thir low weight allow small support construction for the guideway, making it less problematic to construct in urban areas than rails for mass transport systems.

The history of continuous evolution of Personal Rapid Transit (PRT) can be traced back to at least 1953[2]. The 1950s is characterized as pivotal for the car industry, which increased the production of cars dramatically after the American manufacturing economy switched from producing war-related items to consumer goods[3]. As downtowns started beeing dedicated to business, residents were pushed from city centers to suburbs. In stead of using public transport, people working in the city centre commuted by car as a result of reasonable car prices and newly built highways offering a more comfortable and efficient journey between their home and work. This massive increase of road traffic had dramatic effects on the air quality in cities, and the need for a viable compliment to the car was necessary.

While the key features of PRT are small, automatic, private-party vehicles, on-demand service and nonstop trips, the vision of PRT is to interconnect strategic parts of a city centre to multiple suburbs, giving passengers a variety of both departure and arrival stations. Because a pod carries few passengers compared to buses, trains or subways, PRT is not intended for operation between stations with heavy traffic load. On the contrary, PRT is suitable for connecting city centres and rural areas

where the public transport service is less frequent. It might just as well transport people from a mass transport station to a PRT station close to their home, saving them from the waiting time for a bus or a walk. Since PRT arrives at a station on-demand, people don't have to adapt to schedules, which gives PRT customers the same freedom as cars or taxies. In addition, the pods will always choose the shortest route between stations, which is not an option in mass transport system like buses or trams, where fixed routes are followed to increase the covered area. Because the guideways are separated, rush hour traffic will not interfere with the pods, which has the potential of making PRT a more attractive alternative than personal cars.

Currently only two systems categorized as PRT are operational. Heathrow T5 (ULTra), in London, was opened in June 2011 and connects Terminal 5 with the business passenger car park. The system consists of 21 pods, 3 stations and 3.8 km guideaway. The system offers an on-demand service and small personal pods, however only 3 stations does not fulfill the vision of PRT. Skycube(Vectus) in Suncheon Bay, Korea was made operational in April 2014. The system consists of 40 pods and 2 stations. With one station less than the Heathrow T5, neither this can be categorized as a true PRT system.

Unfortunately, the list of failed, abandoned or conceptual PRT projects is much longer than the operational ones. Taxi2000 Corporation has developed the skyweb express PRT system which currently only exist as a prototype that can move along a 60 foot guideway. In the 90s, Raytheon licensed several of the Taxi2000 patents and started a project called PRT2000. Even though the initial design had a lot of flaws, the system was believed to be price and performance competitive with other public transport systems[4]. Nevertheless, the project was abandoned as a result of financial troubles. Computer-controlled Vehicle System(CVS) was developed in Japan in the 1070s. Safe operation was demonstrated for a 1 second headway in 1976. However, the project was abandoned because it was declared unsafe under existing safety standards, specifically the brick wall standard [5]. Finally, CyberCab(2getthere) has been operational in Masdar City, Abu Dhabi since November 2010. It consists of 5 stations(3 of them freight only) and 13 pods (3 of them freight only). However, this is only a test system that will not be further expanded because of the cost of separating it from pedestrian traffic[6].

2.3 Lego Mindstorms

The versatile nature of Lego makes it ideal to build a model of a PRT system from scratch. Even though many may consider Lego as toys for children, it offers advanced components that can be used for building robots. In combination with the Lego City train series, I had a basis with a lot of possibilities. However, since I have already

Figure 2.2: Picture of the Vectus(left) and ULTra(right) PRT systems. The picutres are taken from Podaris **??**, PRT Consulting **??**, connect2edmonton forum **??** and Upriser **??**

spent a lot of time exploring the opportunities with Lego, I decided to base my work on the discoveries and pod design in my project report.

There are few limits to Lego Mindstorms range of use. The latest model, EV3, includes enough sensors and motors to build a wide range of robots. In addition, more sensors can be bought separately, making EV3 a powerful tool with the possibility to utilize samples from color sensors, infrared sensors, ultrasonic sensors, gyro sensors as well as touch sensors. Further, a WiFi-dongle can be connected to the USB port making it easy to connect the EV3 to a standard router, other EV3 bricks or even the Internet.

The robot, which can be built from the standard package, is illustrated in figure figure 2.4. It is programmed to locate the remote control via the infrared sensor, position itself with the servo motors powering the tracks and shoot at the remote. If the remote control is moved, the robot will follow it. The program that gives the robot this functionality is provided my Lego. It can easily be modified in the intuitive standard Lego development tool, which uses function blocks connected in a sequential manner.

2.4 Communication Protocols

There are numerous of different communication protocols in use. Some protocols are custom-made for specific systems while others are designed to serve a wide range of applications. The different communication protocols can be partitioned into abstraction layers to be described visually as a stack where each layer is assigned

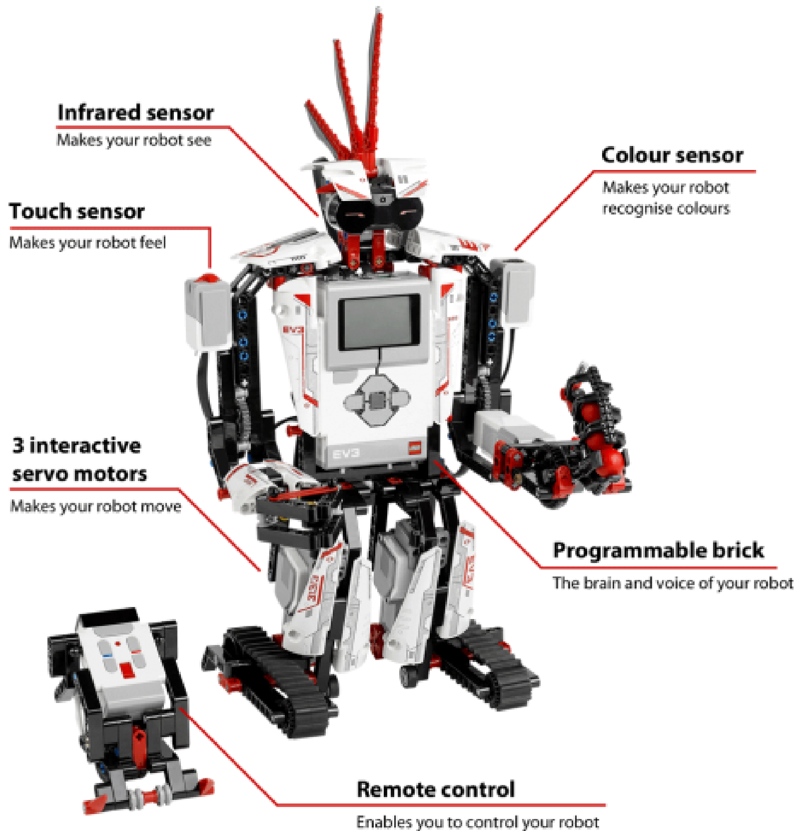


Figure 2.3: The LEGO Mindstorms EV3 Storm Robot. The picture is taken from XmasGiftIdeas [7]

different functionality. A layer serves the layer above it and is served by the layer below it [8].

2.4.1 Transmission Control Protocol

The TCP protocol was initially designed as a highly reliable host-to-host protocol in the U.S. military packet-switched communication network [9]. By the use of sequence numbers and acknowledgements, the TCP protocol guarantees that a stream of data is delivered reliably and in order at the destination [9].

Today, TCP is one of the most common protocols used on the internet today. Because of network congestion, broken links etc, the internet protocol(IP) packets on on the lower layer can be lost, duplicated or delivered out of order [10]. TCP will

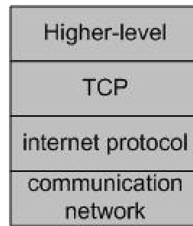


Figure 2.4: Protocol layering. A layer serves the layer above it and is served by the layer below it. The figure is remade from the DARPA protocol specification [9]

detect such situations and correct it to restore the correct order and make sure that packets are received exactly once by requesting retransmission of lost packets and filtering out duplicated ones.

TCP is suitable for applications that require reliable communication not provided by higher protocol layers. An typical example of applications not using TCP is streaming services such as media streaming and gaming where real time and high throughput is more important than receiving each packet in order. If a higher level provides the same functionality as TCP, other protocols such as UDP might be more suitable.

2.4.2 Message Queue Telemetry Transport

MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol[11] implemented above TCP in the protocol layer. MQTT was invented by Andy Stanford-Clark of IBM, and Arlen Nipper of Arcom (now Eurotech), in 1999. It is a publish/subscribe protocol, designed to be light weight and simple to serve units with low computational power interconnected by unreliable communication links. Today it is also used by mobile applications where bandwidth and computational power isn't a problem. An well known application example is facebook messenger, which uses the MQTT protocol to distribute the chat messages[12].

To receive messages, a unit initially subscribes to one or more topics. When a message is published to this topic, the subscriber receives the message. Multiple units can subscribe or publish to the same topic, which means that it supports broadcast. Each message is published to a specific topic. One unit can subscribe to multiple topics at the same time. The distribution of messages are handled by a broker, which is the only unit all subscribing and publishing clients are in direct contact with. The topics are organized in a hierarchy with the different levels separated by "/". In a smart house system, a typical hierarchy could be

smarthouseRoad21/garage/temperature/t1

smarthouseRoad21/garage/door
smarthouseRoad21/masterBedroom/temperature/t2

To administer multiple topics, wildcards can be used:
"+" is a wildcard for a single topic level
"#" is a wildcard for all remaining levels of a topic.

To subscribe to all the temperature sensors in the house, the topic can be set to smarthouseRoad21/+/temperature/#. To subscribe to all the sensors in the garage, the topic can be set to smarthouseRoad21/garage/#.

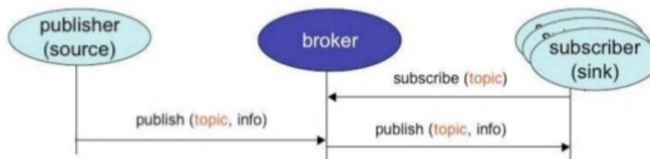


Figure 2.5: A simplified illustration of the MQTT publish/subsrcibe procedure. The picture is copied from [13].

A message might be set to *retained*. Retained messages will be stored by the broker. When a new subscriber matches the topic in the retained message, the message will be sent to the subscriber. This is a useful way of distributing messages that is updated infrequently because the new subscriber will immediately get the message instead of waiting for the next update.

2.5 Lego Java Operating System

Because the standard development tool for EV3 provided by Lego is intended for everyone to understand, its functionality is limited. To make it possible to use Lego Mindstorms for more advanced programs, the firmware can be replaced with leJOS, which includes a Java virtual machine. The LeJos project was started in late 1999 by José Solórzano as a hobby open source project[14]. In the following years, many contributors have joined.

Currently, the latest version supports Java 7 Standard Edition, which is a major update launched on July 7, 2011[15]. This increases the range of use drastically because of all the supported Java standard libraries. In contrast to the standard firmware, it also makes WiFi-communication possible.

Numerous of projects have been made by using Lego EV3 in combination with leJOS. My favourite is CUBESTORMER 3, which holds the current Guinnes World Record for solving a Rubik's cube in 3.253 seconds. In this project, a Samsung

Galaxy S4 smart phone analyses the cube and sends commands to the Lego EV3 bricks which are used to control the motors that manipulate the cube. The video is published on Youtube [16].

2.6 Reactive Blocks

Reactive Blocks is a tool for developing Java programs in a graphical interface by designing blocks based on UML activity or state diagrams. The visual program design is done by adding and connecting blocks to combine the functionalities offered by each block. Blocks with basic functionality can be made entirely of visual components. However, several lines of code is usually needed to supply these components.

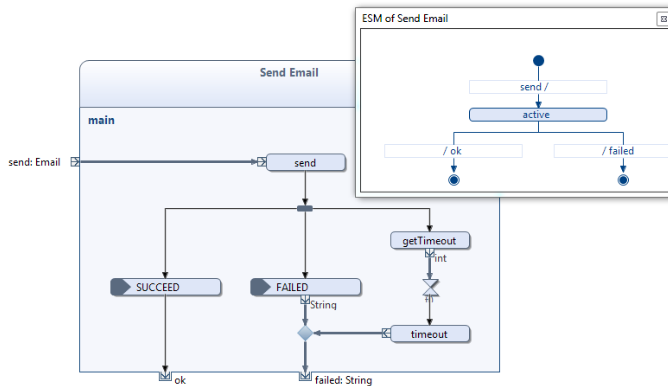


Figure 2.6: An example of one of the included building blocks in Reactive Blocks. The activity diagram is on the left and the belonging external state machine diagram on the left. The same example picture can be found on Bitreactive.com [17].

By combining the blocks included in libraries with own blocks, it is possible to create a new higher level block with added functionality. Essentially, this can be compared to creating a new programming interface(API) with the necessary information visually described in the external state machine diagram figure 2.6.

To use a block, it can simply be dragged and dropped into other blocks to form an intuitive hierarchy. If a block is connected to other blocks in a way that violates the external behaviour of a block, the tool will return an error and illustrate the problem visually. An example where a logical error in the program is revealed by the included analyse tool is illustrated in figure 2.7.

2.7 Project at NTNU

In the previous semester, I wrote a project report where I briefly explored the capabilities of using Lego Mindstorms to simulate PRT systems. I concluded that

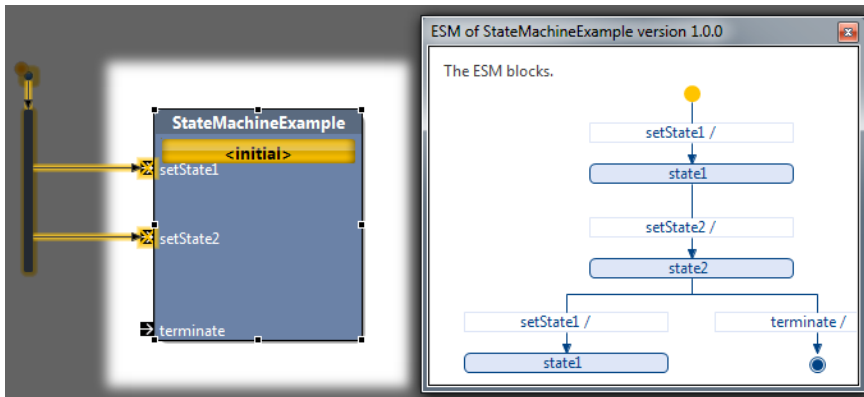


Figure 2.7: I try to activate both states in the state machine example block at the same time (left). By using the included analysing tool, the framework reveals that this behaviour violates the external state machine diagram (right).

it is posible to make the EV3 bricks cooperate as embedded units in a distributed system via WiFi communication. I am reusing the pod design in the current system as well as the switch operators making it possible to automatically switch the pods in intersections.

Chapter 3 Initial Analysis

This chapter describes the decisions and analysis I did before the process of designing the system was started. The choices are substantiated by a brief argument.

3.1 Choosing ITS Application

I decided to focus on Personal Rapid Transport in this project for two main reasons. First, PRT combines a lot of different ITS applications. Initially, some alternatives to PRT was an automatic train stopping system or simulating in-vehicle signage at light regulated intersections. However, PRT use both of these applications to control the pods in an efficient and safe way. Secondly, I experienced that it takes a lot of time to physically design a vehicle, even from Lego bricks. Using the physical design solutions from the previous project saves a lot of time. In addition, reusing the Lego parts avoids the time consuming process of ordering new parts. Finally, this solution keeps the cost at a minimum.

3.2 System Requirements

Before designing a system, it's important to have a clear understanding of what the system should actually do. It is also useful to prioritize what the most important functions are in case a tradeoff situation occur where the quality of one function has to be reduced in order to make another function work properly and vice versa. This is usually formalized as functional and non-functional requirements.

3.2.1 Functional Requirements

Functional requirements describes *what* the system should do. I have chosen the following three main requirements.

1. No crashes - the pods must be controlled in a way that makes it impossible for them to crash into each other.
2. Shortest path - The pods must be routed the shortest possible route between stations in order to maximize the utilization.
3. Easy configuration - after redesigning the guideway, as few parameters as possible should be entered into the system in order to make it work.

3.2.2 Non-functional Requirements

Along with functional requirements, non-functional requirements are also an important aspect which needs to be considered before starting the system design process. Non-functional requirements describe how the system should behave.

1. Scalable - The system must perform even if it is expanded.
2. Smooth behaviour - the pods must start and stop in a natural way. This imply that when the pod approaches a station or stop point, it must decelerate evenly to a complete stop.
3. Avoid stopping on the transit track - as long as the pods are on the transit track, they should move as long as there are no obstacles ahead. At merging intersections, pods on the transit track have higher priority than pods on a station track.

3.3 Using Reactive Blocks and MQTT

To avoid a large and complex code base I decided to use Reactive Blocks to help me organize the code in an intuitive hierarchy. Based on experience from previous courses at NTNU where I have used Reactive Blocks, I know that I will save a lot of work because programming and planning can be done in the same step. Finally, Reactive Blocks offers several different MQTT-blocks. This means that I don't have to implement a communication protocol on my own, which can be time consuming and out of the scope of this report.

3.4 Choosing Distributed and Asynchronous Control

Distributed and asynchronous control are usually a natural consequence of each other and the opposite of centralized, synchronous control. The latter option is desirable to optimize the performance under normal operation, but makes the system complex and vulnerable to unforeseen events. Such events might be obstructions in the guideway, such as halted pods or other objects or passengers using long time

to embark the pods at stations. With a large system containing a lot of pods and stations, the probability of such an event occurring is increasing. With a synchronous control system, this would disturb the whole system and halt it. To compensate for unforeseen events, a fallback asynchronous system is anyway needed.

At the same time, an expansion of the system will increase the computational load in a centralized system. With a distributed system, this problem will be solved by adding more independent control units to divide the load. Finally, a distributed system does not rely on one centralized component, reducing the consequences of a failure.

Chapter 4
Physical Components

This chapter explains the physical components included in the system. It has two main sections. The first part explains the different parts of the guideway and how the automatic switching of the pods is handled. The second part describes the pods design and the components taking care of the automatic control of them.

4.1 Building a Guideway From Lego Train Tracks

The guideway consists of Lego City tracks and is keeping the pod in place, i.e. making sure it's travelling along it's intended path. To make the pod able to travel along different paths, automatically operated switches are placed along the guideway to switch the pod in the correct direction. To decrease the complexity, thus increasing the safety, of the system, the pods are only allowed to travel along the guideway in one direction.

4.1.1 Guideway Layout

The guideway is designed to be as similar to real world systems as possible. It consists of station tracks(figure 4.2a) and transit tracks(figure 4.2b). To make the system more efficient, the stations are separated from the transit track, which makes it possible for pods to stop at stations without blocking passing pods. The system consists of 7 stations. Each station track is long enough to hold at least one pod.

Intersections splitting one track into two are called *switching intersections.* To alter between the two splitting tracks, a motor is connected to the yellow switch handle. These are the only active components in the guideway. The intersections joining two tracks into one are called *merging intersections.* These intersections do not require any interaction to merge the pod. If you disregard the motor connected to the switch, there are no physical differences between switching and merging intersections other than their orientation relative to the pods direction of speed.

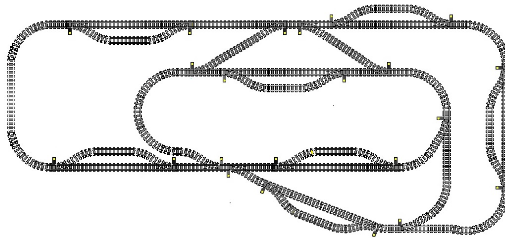
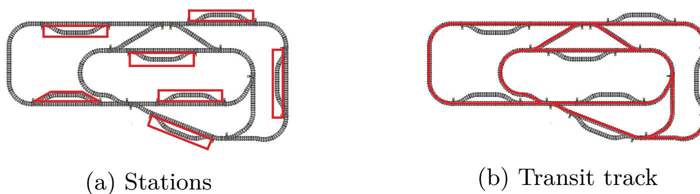


Figure 4.1: The final guideway layout. The guideway consists of 7 stations. Pods are only allowed to travel counter clockwise

To enter a station, the pod needs to be switched into the stations track from the transit track. When exiting the station, the pod will normally merge into the transit track. However, the station track can also end in a switching intersection. This could be useful if a station is expected to have a high traffic load. In this case a loop can be formed from a switching intersection at the end of the station track to a merging intersection at the start. This loop can be used to store pods until they are needed.



(a) Stations

(b) Transit track

Figure 4.2: The station track (a) makes sure pods stopping at stations doesn't interfere with passing pods on the transit track (b).

There are 11 switching intersections and 11 merging intersections, which come in pairs. 7 of these intersection pairs are connecting station tracks to the transit track. The remaining intersection pairs are transit intersections. These intersections connect the 4 circles forming the transit track. The switching transit intersections will switch the pod towards its destination station. In two occasions, a merging intersection is placed directly in front of a switching intersection. This results in a double intersection, with two entrance tracks and two exit tracks.

The double intersection targeted by the exclamation mark in figure 4.3b is the most critical point in the guideway because no pod can complete a whole round without passing this point. If the track between the two intersections are blocked by a stalled pod, the whole system will stop. If this happens any other place along the guideway, at most 3 of 7 stations are made unreachable. The intersection is also expected to be the most likely merging intersection to cause congestion because all

pods needs to pass this point.

(a) Switching transit intersections (b) Double intersections

Figure 4.3: Switching transit intersections(a) and double intersections(b) are marked in red. Pods are only allowed to travel counter clockwise. The exclamation mark targets the guideways critical point

4.1.2 Automatic Switching of the Pods

To operate the switching intersections, a large servo motor is connected to the switch handle. The motor has two pre-set positions which will either switch the pod straight or to the side. The motor is controlled and powered by a EV3 brick. Each brick supports 4 motors, which means that a total of 3 EV3 bricks are needed to operate the switching intersections.

The motors are connected to a bar transforming the rotary movement into a linear. Because there is a lot of resistance in the switch handle, relatively much force is generated. This requires the motor to be more anchored by Lego bricks than illustrated in figure 4.4. For the same reason, the switch is considered to be the mechanical component most likely to fail unless glue is applied to the Lego bricks.

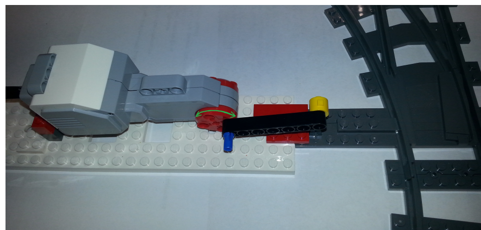
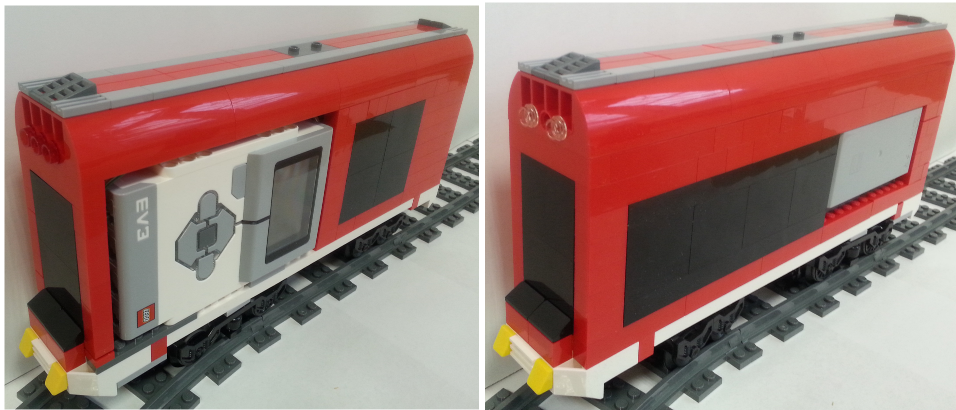


Figure 4.4: The motor operating the switch. The bar transfers the rotary movement into a liner.

4.2 Pod Components

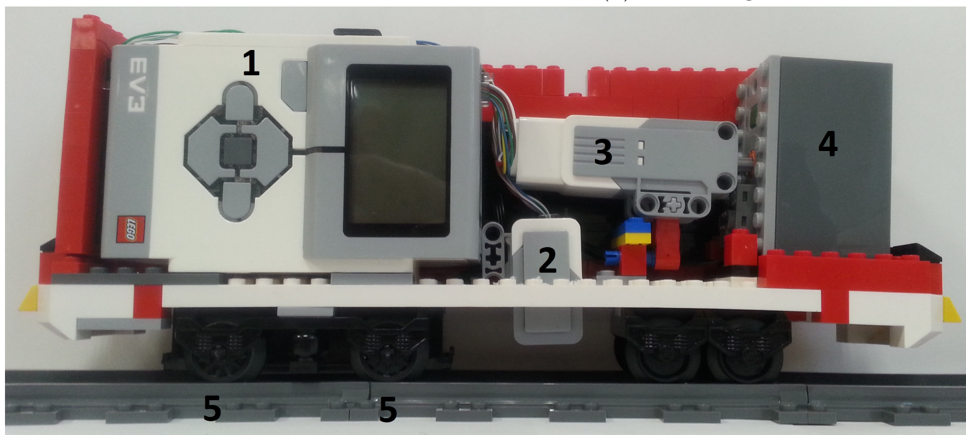
To enable automatic control, each pod is equipped with a EV3 brick responsible for controlling the speed and keeping track of the pods position. The pods are assembled

by parts from a Lego City train.



(a) Pod facing backwards

(b) Pod facing forwards



(c) Pods components

Figure 4.5: Pod from different angels. Figure 4.5c reveals the controllable components of the pod, which is the EV3 brick(1), color sensor(2), speed control servo(3), battery box(4) and propulsion wheels(5)

4.2.1 Color Sensor

The color sensor is directed downwards on the tracks and feeds the EV3 brick with color samples. This makes the pod able to distinguish between the sleepers and table visible between the sleepers. Since the color sensor can recognize 7 different colors, the sleepers can have different colors assigned different functions. The sample rate of the color sensor is 1000, which means that it detects a color 1000 times per second.

4.2.2 Propulsion Control

To change the speed of the pod, the voltage fed to the motor running the propulsion wheels is regulated by the battery box. To operate the battery box, a 15 step rotary switch in front of it needs to be turned. In the center position, the propulsion wheels doesn't move. Turning the switch one way will increase the speed of the propulsion wheels in one direction while turning it the opposite way will make the wheels spin in the other direction. There are 7 levels of speed in each direction. Because of the high centre of gravity, the pod will tip over in corners if the speed level is set higher than 5.

Because Lego Mindstorms and Lego City components aren't compatible, controlling the pods speed require a mechanical interface between the EV3 brick and the battery box. To change the position of the rotary switch on the battery box, the EV3 brick needs to use a medium servo connected to it with a bar. To make sure the intended speed level is set every time, it is important that the servo and battery box cannot move relative to each other.

After looking at some forums online, it seems like it's possible to connect a third party infrared sensor to NXT bricks, which is the predecessor of EV3, capable of emitting the same signals as the standard Lego remote control. If this would work on EV3 as well, the remote controlled voltage regulator included in the Lego City Train set could be used instead of the speed control servo. However, I couldn't find any documentation of this being possible and decided to use the mechanic interface.

Chapter 5

Testing the First Version of the Pod Control System

This chapter briefly explains the testing of the first version of the pod control system and the three main problems that occurred. Each problem is analysed before a solution or possible solution is described.

5.1 Initial System Design

When programming the first edition of the pods control system, I based the work on the system design I used in my project. To keep track of the pods, this system divided the guideway in segments separated by red sleepers. Every time a pod detected a red sleeper, it would check if it had permission to cross the next one. If not, it would stop. Such permissions were given automatically by the system based on the location of the other pods along the guideway. Two pods were not allowed to be in the same segment, which meant that a pod could not be allowed to enter a section if it was already occupied by another pod.

After building a pod and implementing the program controlling it, a number of tests were run to check the performance. These tests revealed the following problems with the pod:

- The color sensor is only detecting blue segment dividers.
- The color sensor detects the same segment divider multiple times while passing it.
- The stopping distance is too long.

5.2 Only Detecting Blue Segment Dividers

When passing black, red or white sleepers, the pod didn't detect them. Initially I thought this was caused by a defect color sensor, but changing the sensor didn't solve the problem. Accordingly, I suspected that the distance between the sleeper and color sensor was too big. I tried to build the sleepers higher to reduce the distance.

Especially when doing this with black sleepers, the pod detected it sometimes as red. When trying blue sleepers, the pod registered it without a problem regardless of the distance. I quickly realized that the problem had to be how the pod interpreted the signals from the color sensor.

The color sensor checks the color each millisecond and returns an integer, called a colorID, representing a color. To make the code readable to humans, I made seven instances of integers. Each reference to the integer was given the name of the corresponding color. It turned out that the relations I had programmed between the colorIDs and colors were wrong. After making a program that printed the detected colorID, I could find out which color each colorID represented by holding LEGO bricks with different colors in front of the color sensor and writing down the colorID printed on the screen. It turned out that blue was the only color that had a correct relation to the colorID. After updating the relations, sleepers in all colors were detected correctly without a problem at all speeds. My guess is that black sleepers were registered as red because holding it close confused the color sensor to return another colorID, which randomly was related to red at the time.

5.3 Detecting the Same Segment Divider Multiple Times

When the color sensor detected the colors, the opposite problem occurred. Now the same sleeper was detected too many times. I had already tried to avoid this by programming the pod to only register a sleeper when the color sensor detected a color change to for example red from another color. It turned out that the sample rate of the color sensor is actually too high. This results in inconsistent readings in the transition between colors, which could result in the color sensor to return up to three color changes instead of only one in the transition between two colors. The solution was to put the color sensor thread to sleep after detecting a color change. After testing, 5 ms was the proper sleeping time for speed levels 3 to 5. Shorter sleeping time didn't remove the problem while longer sleeping time caused the color sensor to miss some sleepers at high speed. At speed level 2, which is the slowest, 5 ms was too short. When the pod is travelling at speed level 2, the sleeping time must be increased.

5.4 Too Long Stopping Distance

When the color sensor worked properly, the program controlling the pod was behaving as it should. All sleepers were detected only once without exceptions. But it took the pod around 65 cm to stop from the maximum speed, which was much longer than expected. With the planned system design, this would cause each segment to be close to a meter long when the length of a pod plus safety distance were added to the stopping distance. Considering that one pod is 32 cm and occupies at least

one segment at all times, this would cause the capacity to be roughly 1/3 of the theoretical capacity since one pod occupies at least the same stretch as the length of three pods. Keeping in mind that a moving pod occupied two segments, only 1/6 of the capacity would be exploited if all pods were moving. In order to make a realistic representation of a real world PRT system, the stopping distance needed to be reduced. The stopping distance is given by the formula

$$D_{total} = D_{reaction} + D_{braking} = vt_{reaction} + \frac{v^2}{a} \tag{5.1}$$

D = distance
v = speed
t = time
a = deceleration

Except for reducing the speed, there are two variables to look into - deceleration and reaction time.

5.5 Braking distance

Because there is no dedicated brake system on the pods, the forces affecting the deceleration was initially only the friction between the wheels and guideway and the friction inside the propulsion motor when no power was applied. By setting the speed to a negative value, it is possible to increase the friction inside the propulsion motor drastically. Setting the speed to -2 is the most optimal setting because -3 makes the wheels spin, decreasing the deceleration. The stopping distance was now reduced to around 40 cm. When the pod started braking, it made a different sound than when running normally. From observing the distance covered before and after the pod made the braking sounds, it seemed like the reaction distance was now significantly longer than the braking distance.

5.6 Reaction Distance

Reaction time is the elapsed time from the color sensor passes a sleeper until the speed on the battery box is set to -2. This time will be referred to as delay and can be divided in software delay and hardware delay. I consider hardware delay to consist of signalling delay and motor rotation time. Signalling delay is the time it takes the color sensor to read a color sample, transfer the signal to the intelligent brick and for the intelligent brick to notify the program. The time it takes from the program orders the motor to rotate until the motor starts moving is also categorized as signalling delay. Rotation time is the time it takes the motor to switch the speed on the battery box, and is the most obvious hardware delay. Software delay is the time it takes from the program receives the color sample until it orders the propulsion

control servo to rotate the motor. This includes the processing time in program, which is probably the most conspicuous software delay.

I set up some different tests to be able to pinpoint where the major delays were located. The first test was to check the signalling delay from the color sensor to the program. To do this, I programmed the intelligent brick to change color of the leds when a segment divider. By using a high speed camera, I could determine how long time it took from the color sensor passed the segment divider to the leds changed color. It turned out that this time was around 40 ms, which corresponds to a distance of 3 cm.

The second test was to measure the rotation time of the propulsion control servo. After reading the LeJOS API documentation thoroughly, I noticed the option "immediate return" when sending commands to the servo. This option decides whether or not the program should wait for the rotation to complete before executing the code on the next line. Immediate return is set to true if not specified. When setting it to false, it was possible to measure the rotation time by logging the time immediately before and after the servo command and calculate the difference. The results were shocking. The rotation time from speed 5 to -2 was between 338 ms and 365 ms. The maximum speed given by the LeJOS documentation was approximately 900 degrees/s, which should result in a rotation time of $t_{rotation} = \frac{105°}{900°/s} = 116ms$. However, acceleration and deceleration of the rotary movement has to be added to this time. When I tested the rotation time without load, i.e. when the servo was disconnected from the battery box, it took around $190ms$. This is reasonable considering signalling delay, acceleration and deceleration. The reason that the rotation time is doubled when the servo is connected to the battery box is a lot of resistance in the voltage regulator wheel.

Since the cable connecting the motor to the intelligent brick was customized and not a standard LEGO product, I wanted to check if the cable played a part in the rotation time. This was with resistance in mind because it reduces the power delivered to the motor. The resistance in the cable is given by the formula $R = \rho\frac{l}{A}$ where l is the cable length, A the cross-sectional area of the conductor and ρ the resistivity, which is a measure of the material's ability to oppose electric current. While I knew that the standard cable was longer than the customized one, I didn't know the area of the cables. However, I assumed that the difference in resistivity was insignificant because both conductors were copper. Poorly attached connectors could also affect the resistance. When testing the rotation time without load, the time was approximately $10ms$ longer with standard LEGO cables than customized cables. It is hard to conclude what causes the difference, but if the area and resistivity are the same for both cables, the results make sense because the customized cable is much shorter than the standard one.

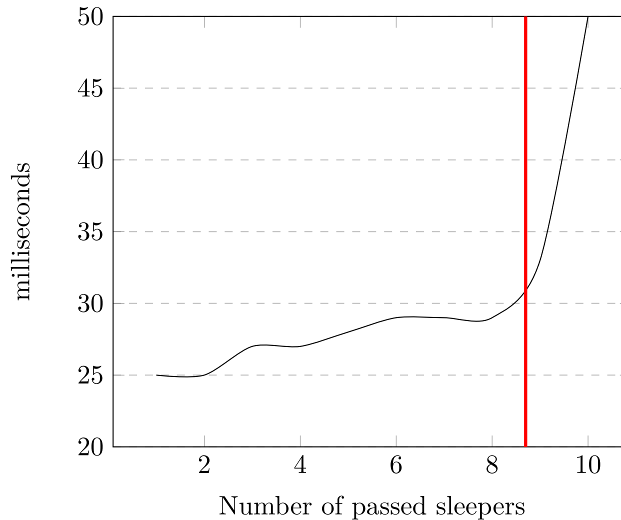


Figure 5.1: Reaction distance vs braking distance. The red vertical line represents the distance covered when the motor is set to speed level -2.

5.7 Reaction Distance vs Braking Distance

For the next test, I wanted to check how long the reaction time was compared to the braking distance. By making the color sensor detect each sleeper in the guideway, i.e., not only the coloured ones, it was possible to log how many sleepers the color sensor passed and how long time it took to pass each single sleeper. By plotting these results into a graph, it's possible to see when the pod is actually starting to brake. Figure 5.1 illustrates the time it takes to pass each sleeper. This is done by registering the time difference when the color sensor detects the sleeper and when it detects the table between the sleepers. Sleeper 0 is the sleeper where the pod should start the braking process. Sleeper 10 is the sleeper it stopped at. The red vertical line represents the time when the motor reported the rotation to be finished, i.e. the speed was set to -2. From sleeper 1 to sleeper 8, it takes longer and longer to pass each sleeper, which means that the pod is decelerating. This is probably the distance travelled while the speed level on the battery box is changed from 5 to 0. When passing sleeper 8, the graph shows a significant increase in deceleration. This is most likely when the speed is set to -1. To figure out if the change in graph gradient from sleeper 9 and sleeper 10 mainly was caused by the laws of physics or increase of braking power(change from level -1 to -2), I compared the braking distance when setting the motor to -2 and -1. No measurable results were found, which means that speed level -1 will be used for braking from now on.

5.8 Reducing the Rotation Time

Since the reaction time is dominated by the rotation time, I started to think of solutions to reduce it. One solution could be to make a gear system by using cogs of different size, illustrated in Figure 5.1. By having a large cog connected to the motor and a smaller cog connected to the battery box, the degrees needed to set the speed from 5 to -2 could be reduced. Normally, each speed level require a rotation of 15°. With the gear system illustrated in Figure 5.1, the motor rotation would be reduced to $\frac{15° \times 3}{5} = 9°$ At the same time, the torque needed to rotate the voltage controller on the battery box would be increased by 40%. When building such a gear system, the needed torque was simply to big for the LEGO bricks to keep the servo in place, which made the gear connected to the servo jump out of position and spin without transmitting the movement to the smaller cog connected to the battery box. This made it impossible to control the pod because it was too unreliable.

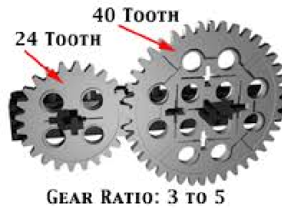


Figure 5.2: LEGO gear

Another solution could be to open the battery box and somehow remove the mechanical parts that adds the resistance between each speed level. If it's possible to remove this resistance completely, it would reduce the rotation time by approximately 45%. In combination with a gear system, which would probably work much better without resistance, the rotation time could be further reduced. A close examination of the battery boxes revealed that they are not intended to be opened. Trying to open them, not to mention modifying them, would include the risk of braking them permanently. I concluded that the possible reduction of reaction time weren't worth the risk and time it would take to make modifications to the battery boxes.

Chapter 6

Speed and Position Testing

This chapter is divided in two parts. First, the color sensors ability to keep track of the position by counting sleepers is tested with different approaches. In the second part, I describe how the color sensor can measure the speed, and present the results from the speed measurements.

6.1 Utilizing the Potential of the Color Sensor

While running the tests on braking distance vs reaction distance in the chapter above, I discovered that the color sensor was much more versatile than anticipated. At first sight, it seemed like it was able to detect every single sleeper and give relative reliable measurements of speed. These are important features because it allows the pods to be aware of it's location with the accuracy of the distance between each sleeper, which is 3,2 cm. To find out how reliable these readings actually were, I set up two different tests. One to make sure the color sensor were actually detecting each and every sleeper and the other one to measure how accurate the speed measurements were.

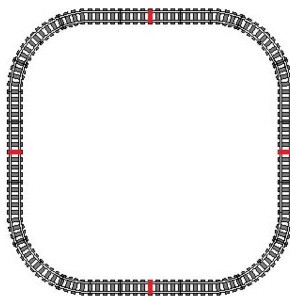


Figure 6.1: Test track. There are 32 sleepers between each red sleeper

6.2 Position Measurement

Since accurate speed measurement is dependent on the color sensor handling the detection of the sleepers correctly, I started testing if it was detecting each sleepers exactly once. I added red sleepers to the middle of each straight side of the circular test track illustrated in figure 6.1 and reprogrammed the pod to return the number of passed sleepers every time it passed a red sleeper. The number should always be 32, but the pod returned inconsistent results, ranging from 28 to 34 sleepers. Changing the sleeping time of the color sensor thread in either direction only made it worse. It was obvious that I needed to try another approach to get more consistent results.

6.2.1 Description of the Color Sensor Thread

To make it easier to understand the following subsections, I am going to describe the central part of the color sensor code in detail. This is necessary to understand the problems and solutions I am describing in the following subsections. The implementation in algorithm 6.1 does the job of detecting coloured sleepers exactly once. However, when registering the normal sleepers, it was inconsistent and giving both to high and low numbers. Line 1 initiates an internal loop. At line 3, the thread goes to sleep until it receives a new color sample from the color sensor, which is stored as detectedColorID. Further, line 4 checks if a color change has happened to avoid reporting the same sleeper multiple times. Line 6 and 7 are repeated for all sleeper colors, but this is not included here for simplicity. Line 7 is the line reporting the detected sleeper to the pod controller. Line 8 checks if the color transition was to a sleeper from the table color, i.e. that a new sleeper is detected. If this is the case, it registers the sleeper by incrementing the passedSleeper variable in line 9. When the pod passes a red sleeper, the passedSleepers variable is registered in an array and reset to 0. Line 11 is the crucial part of the code. This line is supposed to manage the problems occurring at color transitions by putting the color sensor thread to sleep for a given time. X=5 ms was an appropriate value for detecting colored sleepers.

6.2.2 Testing the Color Sensor

To get a better understanding of what the color sensor were actually registering, I started to store each color change in the colorList array and created a small test track. The test track is illustrated in figure 6.2. The pods travelled from the left grey plate to the right one. When pushing a button on the intelligent brick, the array is sent to a running server and printed to the console before it's cleared. During the tests, the propulsion controller motor were disconnected from the battery box to make it possible to control the speed of the pod manually by turning the controller wheel. I

Algorithm 6.1 Original code

```
1  while(true)
2  {
3   detectedColorID = colorSensor.getColorID();
4   if(detectedColorID!=lastDetectedColorID)
5   {
6    if(detectedColorID == RED)
7     sendToBlock("COLOR", "RED");
8    if(lastDetectedColorID == BROWN && detectedColorID != BROWN)
9     passedSleepers += 1;
10  }
11  Thread.sleep(X);
12  lastDetectedColorID = detectedColorID;
13 }
```

was now able to run several consecutive tests without restarting the program, which saved me a lot of time.

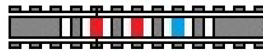


Figure 6.2: Small test track

6.2.3 Checking the Time Difference Since the Last Color Change

My initial idea was to check the elapsed time since the last color change by using *System.nanoTime()*. This approach would make it possible to avoid putting the thread to sleep, which I at the time considered to be the most likely source of the problem. The code is shown in algorithm 6.2. The main change in this code is line 5, where the program will ignore color changes that are more recent than X ms in addition to checking if a color change has happened. To make the condition similar to algorithm 6.1, I initially set X to be 4 ms. The registeredColorID is logged at line 9. Even with such a short test track, the outputs from the tests were to large and inconsistent to understand, and the time difference approach were discarded after some attempts with different values of X.

6.2.4 Require Consecutive Color Samples

My second idea was to require a given number of consecutive color samples to accept it as a color change. The code is shown in algorithm 6.3. Line 4 to 7 are added to count consecutive color samples. When I made the logic in line 8, a new problem

Algorithm 6.2 Checking the time difference since the last color change

```
1  while(true)
2  {
3   detectedColorID = colorSensor.getColorID();
4   timeSinceLastColorChange = (System.nantoTime()/1000000) - lastColorChange
5   if(detectedColorID!=lastDetectedColorID && timeSinceLastColorChange > X)
6   {
7    if(detectedColorID == RED)
8     sendToBlock("COLOR", "RED");
9    colorList.add(detectedColorID);
10   lastColorChange = (System.nanoTime/1000000);
11  }
12  lastDetectedColorID = detectedColorID;
13 }
```

appeared. The logic needed to be changed in order to avoid multiple detections of each sleeper because the lastColorID and detectedColorID always would be the same due to the first condition. To deal with this, I made an extra variable called lastRegisteredColorID. This variable contained the last colorID passed by the check at line 8. By adding the lastRegisteredColorID, it is no longer possible to register the same color multiple successive times. Nonetheless, the transition between colors was still a problem regardless of the value of X. Even with X=20, the program would register one color transition as multiple transitions at slow speeds, while it didn't detect some of the transitions at high speed. The color changes in the returned array were to many to grasp what was wrong and I had to disregard the consecutive color approach as well.

6.2.5 Adding LastRegisteredColorID to the Origial Code

By adding the lastRegisteredColorID to algorithm 6.1 and log each color change, I finally got readable results. However, while a sleep time of 12 ms worked for speed level 3 to 5, speed level 2 required a sleep time of 20 ms in order to not detect the same sleeper multiple times.

The results when using nothing else than the table as sleeper space might look inconsistent at first. But at speed 5, the colors are registered correctly. The inconsistency only appears around the detection of colorID 2, which is blue. At speed 1, the transition between 13 (table color), to 2 (blue) registers the phantom colors 7 (black) and 6 (white). When increasing the speed, the number of phantom colors disappears gradually until the results are correct at speed 5.

Algorithm 6.3 Require consecutive color samples

```
1  while(true)
2  {
3   detectedColorID = colorSensor.getColorID();
4   if(detectedColorID == lastColorID)
5    consequtiveColorIDs += 1;
6   else
7    consequtiveColorIDs = 0;
8   if(consequtiveColorIDs > X && lastRegisteredColorID != detectedColorID)
9   {
10   if(detectedColorID == RED)
11    sendToBlock("COLOR", "RED");
12   colorList.add(detectedColorID);
13   lastRegisteredColorID = detectedColorID;
14  }
15  lastDetectedColorID = detectedColorID;
16 }
```

Detected colorIDs with the table as the layer between sleepers

Speed=1: 13, 7, 13, 0, 13, 7, 13, 0, 13, 7, 13, 7, 6, 2, 6, 7, 13, 7, 13, 7

Speed=2: 13, 7, 13, 0, 13, 7, 13, 0, 13, 7, 13, 7, 6, 2, 7, 13, 7, 13, 7

Speed=3: 13, 7, 13, 0, 13, 7, 13, 0, 13, 7, 13, 7, 2, 13, 7, 13, 7

Speed=4: 13, 7, 13, 0, 13, 7, 13, 0, 13, 7, 13, 7, 2, 13, 7, 13, 7

Speed=5: 13, 7, 13, 0, 13, 7, 13, 0, 13, 7, 13, 2, 13, 7, 13, 7

When using only white as the layer between the sleepers, the problem switched from the blue sleeper to the red ones. Here, color id 13 occur as phantom color relatively often while color id 3 occurs only once in the transition to and from red. Just like when the table is used as padding, the phantom colors gradually disappear as the speed increases. At speed level 5, the results are correct.

Detected colorIDs with white as the layer between sleepers

Speed=1: 6, 7, 6, 13, 3, 0, 13, 6, 7, 6, 13, 3, 0, 13, 6, 7, 6, 2, 6, 7, 6, 7

Speed=2: 6, 7, 6, 13, 0, 13, 6, 7, 6, 13, 0, 13, 6, 7, 6, 2, 6, 7, 6, 7

Speed=3: 6, 7, 6, 13, 0, 13, 6, 7, 6, 13, 0, 6, 7, 6, 2, 6, 7, 6, 7

Speed=4: 6, 7, 6, 13, 0, 6, 7, 6, 13, 0, 6, 7, 6, 2, 6, 7, 6, 7

Speed=5: 6, 7, 6, 0, 6, 7, 6, 0, 6, 7, 6, 2, 6, 7, 6, 7

In stead of trying to fix the problem by adapting the color sensor code, I concluded that it would be more efficient and easy to use the table surface as sleeper space between all sleepers except the blue ones as illustrated in figure 6.3. When running 3 tests at each speed level, the registered colors were correct every time.

When going back to the test track illustrated in figure 6.1, I ran 9 test rounds with at speed level 5, then 3 rounds each on speed levels 4 to 2. At speed level 1, the pod stopped in the corners, making this speed level unsuitable to use and therefore not included in the results. The total number of passed sleepers was 2304. From all these sleepers, only one were missed, which corresponds to a deviation of 0,00043%. However, red sleepers are sometimes interpreted as yellow, which means that yellow and red must be considered undistinguishable.

So far, I haven't been able to test with green and yellow sleepers because I don't have access to flat LEGO bricks that would fit on the sleepers in those colors. Anyway, yellow sleepers can not be used because the pod has problems telling the difference between yellow and red sleepers. However, the tests show that it is in deed possible to detect each and every sleeper at speed levels 2 to 5 with a deviation of 0,00043

Figure 6.3: Small test track with different color between the sleepers

6.2.6 Intersections and Flexible Tracks

The sleepers in straight and curved track sections are now detected because they are all the same length with the same space between them. However, the intersections and flexible track sections differs from the normal tracks as shown in figure 6.4 and figure 6.5. In intersections, the space between the sleepers is inconsistent. As a result, the pod will not know it's exact position inside and when exiting an intersection.

Figure 6.4: Picture of two intersections

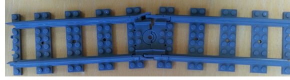


Figure 6.5: Picture of a flexible track between two straight tracks

6.3 Speed Measurement

Now that each sleeper is detected, it was time to run some tests to see if it's possible to measure the speed of the pod. Getting reliable real time speed readings is important to know the actual speed in stead of the assumed speed based on the current speed level on the battery box. There are mainly two reasons why the actual speed can deviate from the estimated speed. Firstly, the speed could be set to another level than intended because the servo motor or battery box is out of position. Secondly, the speed will be reduced when the batteries supplying the rotary propulsion motor is drained as well as in corners because of increased friction.

6.3.1 Speed Calculation

To calculate the speed, the standard formula is used:

$$\bar{v} = \frac{\Delta s}{\Delta t} = \frac{s_1 - s_0}{t_1 - t_0} \tag{6.1}$$

$\bar{v}$ = average speed
s_0/t_0 = start point
s_1/t_1 = end point

t_0 and t_1 are measured by logging the moment certain points in the guideway is detected by the color sensor. s_0 and s_1 are the physical position of these points. It's important that these points have the same distance at each calculation. In the case of the test track in figure 6.1, this is easy to achieve because the track is symmetric. The speed calculation could be done each time the pod passes a red sleeper or every time it completes a lap. However, the track that's going to be implemented in the LeMIP system will not be symmetrical, which means that I need to use sections that will repeat and have the same distance all over the track. Since most of the track is made by straights and turns, the largest section that repeats frequently around the track is one straight or turn section, which has the same length and consists of four sleepers. Since each such section is symmetric, it can be divided in even smaller sections, illustrated in figure 6.6.

To be able to use the speed measurements, it's important that the measured speed is close to the actual speed at any moment. In order to make this possible, the interval between each measurement needs to be as small as possible. At the same time, it's

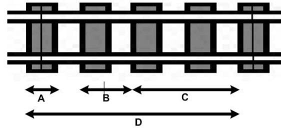


Figure 6.6: Different values of Δs

important that the measurements are consistent. There is a tradeoff between these two requirements. The shorter Δs is, the more inconsistent the measurements will be because each inaccuracy will have a greater impact. On the other hand, longer Δs will be more tolerant for inaccuracies and give more consistent results, but not necessarily the actual speed because a change in speed will take longer to detect. In other words, long Δs is preferred when the speed is expected to be constant. When frequent speed changes are expected, short Δs will be more convenient.

In the LeMIP system, I expect that the speed changes will happen in corners, intersections and stations. In corners, the friction between the wheels and guideway is increased, resulting in reduced speed. At merging intersections, congestion is expected, implying a lot of start and stops, just like driving in rush hour traffic. At stations, a reduction in speed is necessary to ensure accuracy. This is needed in order to stop at the intended point, which is crucial to allow multiple pods at the same station in a safe way.

6.3.2 Selecting the Value of Δs

In order to find the appropriate compromise between instantaneous and consistent speed measurements, I decided to run a series of tests and compare the results between the different values of Δs. To limit the amount of tests and data, I only ran the tests on speed level 5, where the results were the most inconsistent. Using alternative A was not possible because the sleepers and table are of different height relative to the color sensor. This caused the color sensor to detect the color of the sleepers longer than the padding, giving inconsistent and wrong speed measurements.

To test how consistent the speed measurements were with different values of δs, I ran the pod two rounds around the test track. Each round is 128 sleepers. The graphs in figure 6.7, figure 6.8 and figure 6.9 are based on results from the same test run. In all three graphs, it's possible to see that the pod's speed varies between the straights and corners. The first lap had an error reading at sleeper 87 and the second lap a similar an error reading at sleeper 8. With longer δs, the deviation caused by these error readings are less decisive, which presents a more consistent graph.

Let's imagine a scenario where the pod is calculating which sleeper it should start to brake at in order to stop at a certain point. Doing these calculations with a

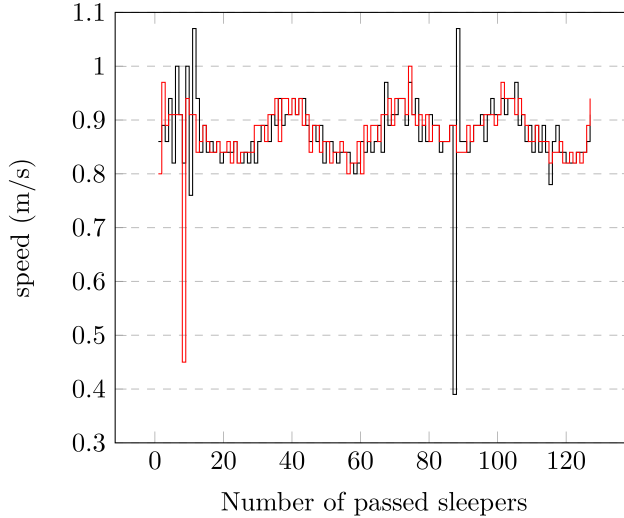


Figure 6.7: Speed with Δs set to alternative B. The pod travels two rounds around a test track. The different rounds are represented by different colors. Ideally, the two graphs should be identical.

lower estimate of the speed than the actual speed would result in a longer stopping distance than the calculated one. If we assume no security distance is added to the calculations, the pod would pass the intended stopping point, which could cause a crash into another pod.

If a pod is travelling along the tracks and an error reading similar to the ones in our test run occurred at exactly the critical sleeper where it should start the stopping process, the different values of Δs will have an effect on how many extra sleepers it passes before stopping. Assuming that there are no consecutive error readings, the pod will only need one additional sleeper with alternative B. This is because the error will be detected at the next sleeper, where the speed measurement is correct. With alternative C, the pod will need maximum two extra sleepers for the same reason. Likewise, it needs maximum 4 sleepers with alternative D. The reason I'm using maximum is because this is worst case scenarios where several unlikely events needs to occur at the same time. The first condition to realize the worst case scenario is that the error reading must occur exactly at the critical sleeper. Another condition is that the deviation must be severe enough to prevent the pod from starting the stopping procedure based on the incorrectly measured speed before the error is corrected. When the pod is passing flexible tracks and intersections, error readings will occur because the sleepers are different in these sections. The lower the value of Δs is, the faster the pod will get correct speed measurements.

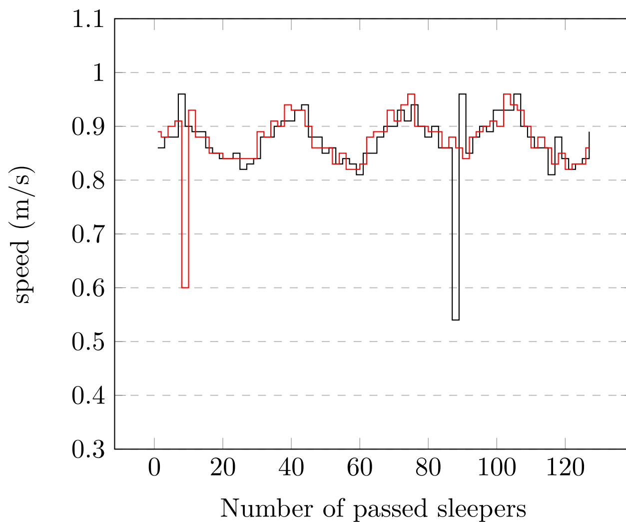


Figure 6.8: Speed with Δs set to alternative C. The pod travels two rounds around a test track. The different rounds are represented by different colors. Ideally, the two graphs should be identical.

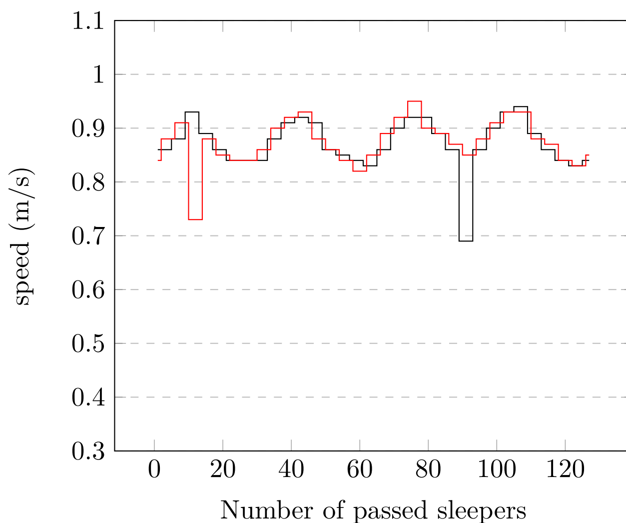


Figure 6.9: Speed with Δs set to alternative D. The pod travels two rounds around a test track. The different rounds are represented by different colors. Ideally, the two graphs should be identical.

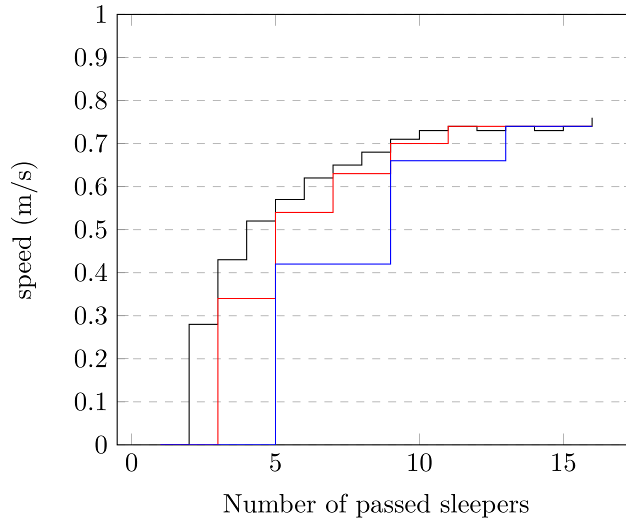


Figure 6.10: Acceleration with all the different Δs alternatives. The black line represents alternative B, red alternative C and blue alternative D.

The graphs in figure 6.10 and figure 6.11 represents the measured speed while the pod is accelerating and retarding. The measurements seems reasonably consistent regardless of the δs value during both acceleration and retardation. On the other hand, the longer δs is, the more delayed the measurements are relative to the actual speed. In addition, using alternative D, represented by the blue line, each speed measurement is wrong because it's the average speed of the last four sleepers. During acceleration this would cause a much longer stopping distance than calculated because the speed always is higher than the measured speed. By assuming that the black line is close to the actual speed, the speed from sleeper 5 to 9 is approximately 35% to 66% higher than the measured speed represented by the blue line. In a worst case scenario this could result in an increase of the stopping distance by more than 4 sleepers because of the severe deviation over a long period of time. During deceleration, the pod would be fooled to think it will not have time to brake normally and apply emergency brakes. The result would be the pod stopping dramatically ahead of where it should, which is undesirable.

To test the speed measurements at the different speed levels, I ran the pod 3 laps at each speed level. The results are presented in figure 6.12. Each speed level is represented by it's own color. From the graph, two important observations can be made. First and most important, there are no occurrences of multiple successive error readings, which means that error readings only causes one extra sleeper to the required safety distance between pods. Second, the different speed levels can be

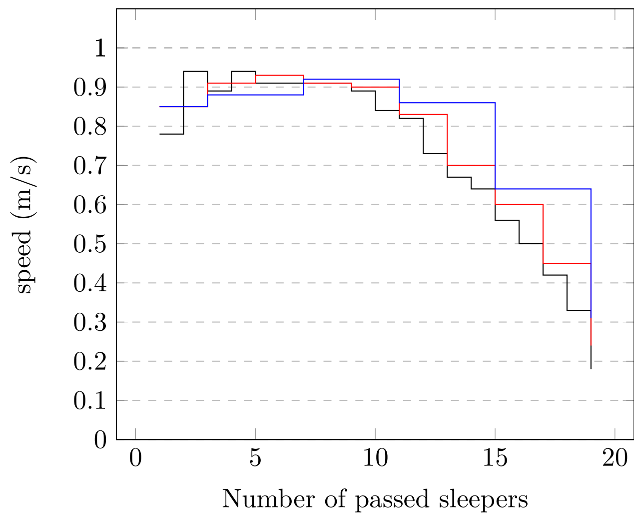


Figure 6.11: Retardation with all the different Δs alternatives. The black line represents alternative B, red alternative C and blue alternative D.

distinguished, which means that the pod can detect a mismatch between the actual speed level and the intended speed level on the battery box.

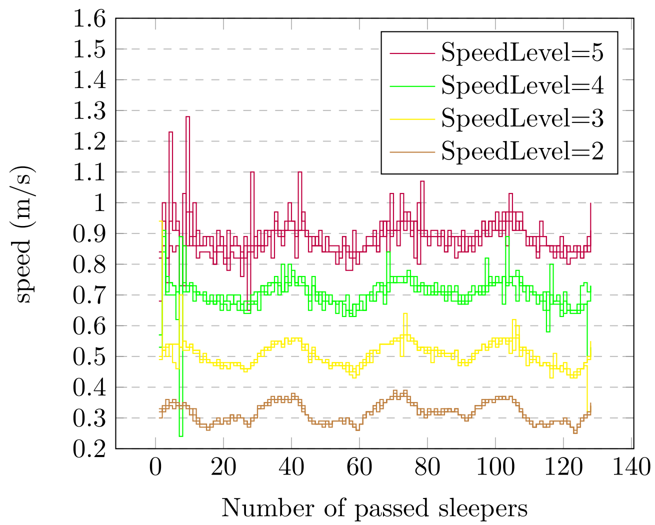


Figure 6.12: Measured speed at speed level 2 to 5

Chapter 7

Stopping and Starting Procedure

This chapter explains how the accurate speed and position measurements can be used to improve the system design. The main focus of this chapter is to discuss and test which approach will stop the pod in the most accurate and smooth way, which is done in the first section. The second section will briefly explain the starting procedure.

7.1 Stopping procedure

I wanted to use the accurate speed and position measurements to improve the pods' behaviour. The original strategy was to control the pods in a *reactive* manner. This means that the pods start the braking procedure when it passes a section divider. Even if the stopping distance was greatly reduced by reversing the rotary motor, the safety distance needed in order to make sure the pods could not crash required a long distance between each section divider, resulting in poor capacity utilization. In addition the stopping process of the pod looked too dramatic to give a realistic simulation of a real world system. Finding a way to control the pods in a *proactive* manner should make it possible to stop the pods in a more controlled way at an accurate stopping point. By *proactive*, I mean that will know when it should start the braking procedure in order to stop at a given point. This serves two purposes. First, the security distance is reduced significantly, allowing better capacity utilization. Second, the pods behaviour will be more responsive and realistic. By running a series of tests, I wanted to determine how close to a specific point it was possible to stop the pods.

7.1.1 Calculate the Stopping Distance by Standard Formula

To calculate the stopping distance, equation (5.1) is needed. However, the acceleration constant and reaction time is still unknown. Based on the results represented in figure 6.11, it's possible to make a rough calculation of the acceleration for speed

level 5.

$$a_5 = \frac{v^2 - v_0^2}{2s} = \frac{(-0.89 \mathrm{m/s})^2}{2 * 0.36m} = -1.1 \mathrm{m/s^2} \tag{7.1}$$

Unfortunately, the acceleration is not constant because the friction in the rotary propulsion motor depends on which speed level the battery box is set to. For example, the forces slowing down the pod is much stronger when the speed level on the battery box is set to 0 than when it's set to 4. Consequently, in order to compute an accurate stopping distance, the current speed level needs to be included in the calculations, which adds complexity. This solution requires the program to make a table lookup in order to find the proper acceleration variable corresponding to the current speed level.

7.1.2 Lookup the Stopping Distance in a Table

Since the table lookup has to be made, I concluded that spending a lot of time on finding the correct reaction time and acceleration for each speed level in order to make the calculations were inconvenient. Building a table with the actual stopping distance seemed more relevant. After studying figure 6.12, I decided to initially divide the table into 8 different rows, two for each speed level, and see how much impact the speed difference have on the stopping distance. Since turns adds friction between the wheels and guideway, the stopping distance will be shorter in a turn than on straights. Because the pod is not aware of where the turns are, the stopping distance needs to be measured on straights in order to account for worst case scenarios.

7.1.3 Calculate the Stopping Distance by Quadratic Function

To measure the stopping distance, I used one pod that would start the stopping process when passing a red sleeper. For each speed level, the sleeper was placed respectively in the end of a turn and approximately 20 sleepers along a straight, as illustrated in figure 7.1. This made it possible to measure the stopping distance at both maximum and minimum speed of speed level 2 to 5. I ran each test set-up 10 times, which gave 80 samples. In addition, I checked the stopping distance at speed level 1, which was 1 sleeper. This was only possible at the straight section because the pod stops in turns at this speed level.

The results from the test is represented in figure 7.2. Each collected data sample is shown as a black dot and looks quadratic at first glance. By adjusting the coefficients of the standard quadratic function, I ended up with the function $31x^2 + 6x$, which seems representable for the data samples. The c coefficient serves as the security distance and is set to 0 because the function corresponds to the worst case scenarios so far.

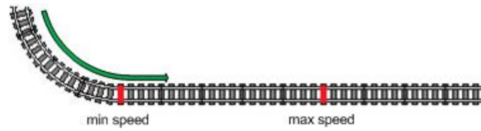


Figure 7.1: Test track for measuring stopping distances

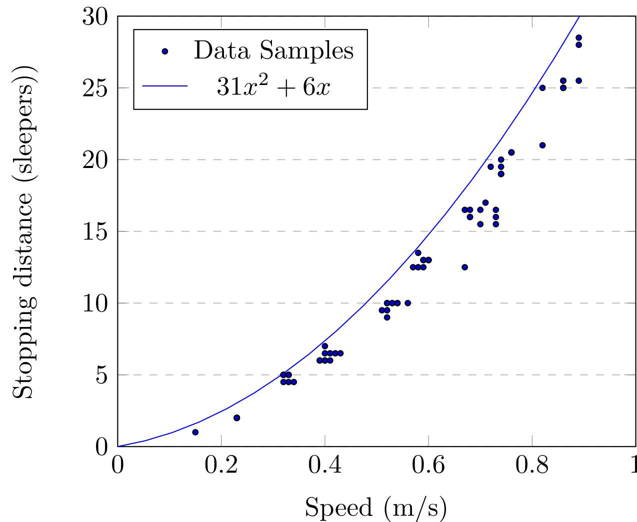


Figure 7.2: Stopping distance at from different speeds

When I tested the function on the pod, the results were pretty good on straights at all speed levels. However, when a turn was included in the stopping distance, the pod stopped way to early. The deviation from the intended stopping point to the actual stopping point increased along with the speed. Keeping in mind that the pod knows it's position with an accuracy of 3.2 cm, a deviation in stopping distance up to 40 cm was unacceptable.

7.2 Avoid Pre-intended Stopping

Pre-intended stopping occurs when the pod stops before the intended position. To avoid pre-intended stopping, I tried two different approaches. Initially, the pod was reprogrammed to detect pre-intended stopping prior to a complete stop by comparing the current speed with the remaining distance to the intended stopping point. If the pod detected an imminent pre-intended stop, it would increase the speed level from 0 to 2 and recalculate the stopping distance. However, after implementation, the pod had a strange behaviour in the stopping procedure. The speed level was switched between 0 and 2 multiple times, causing the pod to bounce the last distance to the

stopping point. Often it would also pass the stopping point, and I discarded this approach instead of figuring out the reason for this behaviour.

My next approach was to increase the security distance and reduce the speed level gradually. I divided the stopping procedure into two steps.

1. Set the speed level to 2 and slow down to the corresponding speed. The initial stopping procedure must be initiated earlier than the previous solution because the resistance in the rotary motor is lower at speed level 2 than 0. To accomplish this, the c coefficient in the formula calculating the braking distance needs to be increased.

2. Set the speed level to 0 when the distance to the stopping point is 4 sleepers.

At straights, this procedure results in the pod stopping exactly at the intended stopping point unless the pod passes any intersections or flexible tracks while the speed level is set to 0. In turns, the pod will stop approximately 2 sleepers before the intended stopping point. It would be possible to compensate for this, but an accuracy of 2 sleepers is more than good enough.

7.2.1 Detect Stop

Because the speed measurements done by the pod is average instead of instantaneous, it's impossible to measure a speed of 0, even if the pod is at a standstill. It's possible to measure a speed close to 0, but when the pod stops, it will still wait for the next sleeper before calculating the next speed measurement. Consequently, if the pod does not detect a new sleeper within a given time, the pod defines it as a complete stop. To make it less likely to misinterpret an error reading as a stop, the pod will only expect a stop when the speed level is set to 0 and the last speed measurement is lower than a certain value. After studying the results in section 6.3, I concluded that if the last speed measurement is lower than 0.5 m/s, the pod will interpret it as a stop if it's more than 0.5 seconds since the last sleeper was detected. These values should take consideration of intersections and flexible tracks, where the sleepers are much longer, preventing the pod from reporting standby before it has come to a complete standstill.

In rare cases, this method of detecting a stop fails, which require an additional stop checking mechanism. When the pod stops with the color sensor directly above a color transition, i.e. the edge of a sleeper, the color sensor will detect the two colors randomly, fooling the pod to believe it's still moving. When situations like this occur, the reported false speed is always relatively high to what the expected speed should be. Thus, if the speed level is set to 0, the last speed measurement is lower than 0.5

Speed Level	Sleep time
1	30 ms
2	20 ms
3	12 ms
4	8 ms
5	8 ms

Table 7.1: The time period the color sensor goes to sleep every time it detects a color change

m/s and the current speed measurement is more than 0.05 m/s higher than the last speed measurement, the pod will interpret it as a complete stop.

However, when implementing these checks, the run time of each iteration in the color sensor while-loop was increased, demanding a change of the sleep time. After running some tests on the different speed levels, I picked the values of Δt presented in table 7.1. Since the rotation time of the speed control motor adjusting the speed level on the battery box is relatively long, Δt will not be changed until the motor has reported that the movement to be finished

7.3 Starting Procedure

When setting the speed level directly from 0 to 5, the sleep time in the color sensor thread will be too short compared to the actual speed, resulting in error readings. To avoid this, the speed will always be incrementally increased with 1 second brake between each speed level until the given maximum speed level is reached.

Chapter 8

Abstract Guideway Design

This chapter is divided in 3 parts. The first part explains how the system keeps track of the pods. The second part describe how the guideay is divided logically to not mix the pods positions. The final part explains which parameters need to be entered manually into the system every time the guideway design is changed.

8.1 Accurate Position Awareness

Since the pods has no anti-collision sensors, the system needs to be aware of the pods accurate position at all times to keep the pods from crashing. Because the pods are running along tracks, the position only needs to be one-directional. The color sensor will detect a sleeper every 3.2 cm, which is the only method used to determine it's position. However, these sleepers all look the same, which requires a way to distinguishing them and give each one an unique identity. To do this, the pods control unit will contain a sleeper counter, which will increment every time a new sleeper is detected. Unfortunately, this alone isn't an adequate solution because the pods drive in circles of different length around the guideway. As a result, each sleeper could have numerous of different numbers based on the previous route of the pod. To not be dependent on knowing the pods route history, the counter needs to be reset in a way that assigns one sleeper only one possible number, i.e identity.

8.2 Dividing the Guideway in Number Series

To be able to assign each sleeper exactly one unique number, the guideway is divided into number series. If the guideway only consists of one track, one number series is sufficient. Since the guideway is circular, the number series has to be reset at one point along the track to avoid overlapping sleeper numbers. For each intersection pair, one additional number series is needed to make sure two sleepers aren't assigned the same number id. As a result, each station track will have it's own number series. Similarly, with the chosen guideway design, the transit track needs to be divided

into at least 4 number series. In figure 8.2a, the guideway is divided into 11 number series, which is the lowest number possible with 11 intersection pairs.

8.2.1 Sleeper Number Assigment in Intersections

Number series can only change in conjunction with intersections. This is realized the same way in switching and merging intersections, as illustrated in figure 8.1. The red sleepers mark the first sleeper in a number series. The red sleepers separating the number series needs to be placed in a way that makes it impossible for pods in different number series to crash. If a pod merging from the side has passed sleeper number 14, the system will deny a pod approaching the merging intersection from the other track to pass sleeper number 13. Similarly, if the pods back end is clear of sleeper number 29, the system knows it can switch the next pod straight without risking a crash because it cannot detect a pod further ahead in the same number series.

In both splitting and merging intersections, the new number series starts at the first separated sleeper marked with red color. Physically, the merging and splitting intersections looks symmetric. Logically they are not. To make them logically symmetric, the red sleeper should be placed one sleeper closer to the merging intersection. The solution with physical symmetry results in one more sleeper with common sleeper id in merging intersections compared to switching intersections, which isn't necessary. The reason is simply a result of the physical design (shown in figure 6.4) of the Lego tracks, which does not allow bricks to be attached to the 3 last sleepers in front of the track shifting plate.

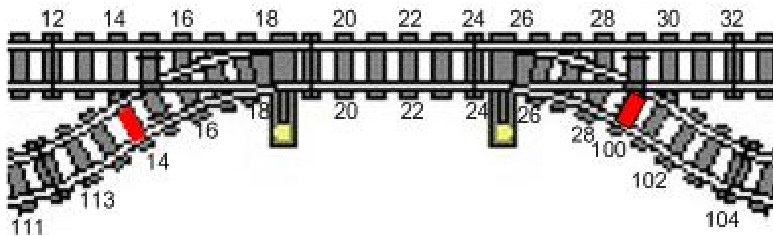


Figure 8.1: The division of number series is the same in switching and merging intersections. The red sleepers mark the transition between number series and is the first sleeper in the new number series. Sleeper number 18 and 26 looks like they consist of two sleepers, but because of no space between them except outside the rails, the color sensor is not able to separate them.

8.2.2 Adjustment Sleepers

The red sleepers separating the number series are called adjustment sleepers. When a pod passes an adjustment sleeper, it recognizes it by the red color. Prior to passing it,

the pod is given the number of the adjustment sleeper by the system. When detecting the adjustment sleeper, the pod sets the sleeper counter to the given number.

8.3 Correction Sleepers

To compensate for error readings, correction sleepers can be placed at locations where the counter is likely to be offset. Contrary to adjustment sleepers, the pods does not need to know the identity number of each correction sleeper. This is solved by using a modulo operation, which finds the remainder after division of one number by another. Donald Knuth describes the modulo operation by the use of floor division with the formula $r = a - n * \frac{a}{n}$. In computer programming, the modulo function is implemented by using the symbol %, i.e the Knuth formula can be simplified to a % n. Some simple examples of the modulo operation are 8 % 10 = 8, 10 % 10 = 0, 12 % 10 = 2.

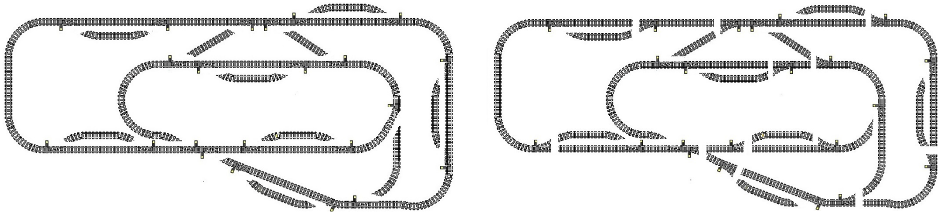
If correction sleepers are used, it is crucial that they are placed at correct sleeper numbers, i.e a sleeper number where the modulo operation will return 0 if the assumed position is correct. However, there is no need to place a correction sleeper at every 9th sleeper, but it should be used after areas likely to cause error readings. Such areas are intersections and at long stretches without adjustment sleepers. If a pod travels along the longest number series in figure 8.2a, it will pass a lot of intersections without any adjustment sleepers. This will most likely cause the sleeper counter to be dangerously offset. Placing one correction sleeper after every intersection would correct such error readings before they become dangerous. If the recommended amount of number series is used, an adjustment sleeper will be placed directly after switching intersections. This will limit the need for correction sleepers to long stretches and directly after merging intersections.

To exemplify how correction sleepers work, let's set n to 9. If a pod passes a correction sleeper assuming that the position is 12, the modulo operation will return 3. The pod will then know that the assumed position is 3 sleepers offset and set the counter to 9. If the assumed position is 26, 8 will be returned. In this case, the counter will be set to 27, not to $26 - 8 = 18$ because the returned number is greater than $\frac{9}{2} = 4.5$.

The advantage of using correction sleepers is that the pods doesn't need to know any other variable than n prior to passing a correction sleeper. Since n is the same for all correction sleepers, it can either be included in the pods control program or given by the system when the pod is started.

A disadvantage is that correction sleepers will worsen the situation if the sleeper counter is offset by more than $\frac{n}{2}$. To avoid this, n must be set to a high enough

value. The higher value of n, the safer it is. On the other hand, an increase of n reduces the options where to put the correction sleeper, which might not make it possible to place it where it's needed.



(a) Minimum number series division (b) The recommended number series division.

Figure 8.2: The space between the number series are only for illustrative purposes. Each station track is assigned one number series. Even if the transit track only needs to be divided in 4 number series(a), it is recommended to have two number series per intersection pair to minimize the necessary renumbering of sleeper ids when adding or removing sections(b).

8.4 Initial Parameters

To make the system aware of how the tracks are put together, it needs to be assigned some initial parameters it can use to number the sleepers and know which number series are connected. To make this possible, every intersection needs to be entered manually into the system.

Each intersection is assigned 3 sleeper numbers.

1. Common sleeper number
2. Side sleeper number
3. Straight sleeper number

The system will also need to know if the intersection is a switching intersection or a merging intersection.

In merging intersections, the common sleeper number is the first number in the intersection that is shared by both tracks, i.e the same number as the adjustment sleeper. In figure 8.1, this is sleeper number 14. The side sleeper number is the number it would be given in the previous number series, which is the last sleeper number plus one. In the illustration this number is 115. Usually, the straight sleeper number is the same as the common sleeper number. However, it is allowed to change the number series on the straight track prior to a merging intersection, just like on the side track. This would require an adjustment sleeper on the straight track as well. In this case, the same rule apply as for numbering the side sleeper.

In switching intersections, the common sleeper number is the number it would be given in the previous number series plus one. In the example this number is 29. The side sleeper number is the first number in the new number series on the side track, i.e the same number as the adjustment sleeper, which is 100 in the illustration. Unlike merging intersections, the straight sleeper number is usually different from the common sleeper number. This requires an adjustment sleeper on the straight track as well as the side track.

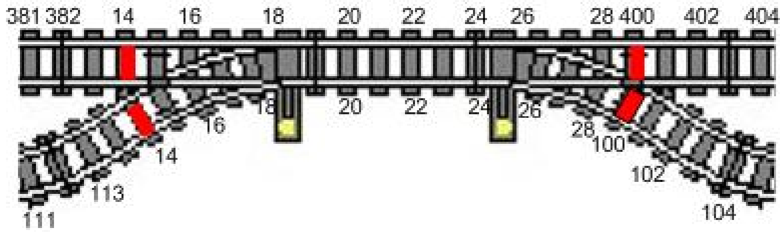


Figure 8.3: Each intersection is connected to 3 different number series. If this is done in all intersections, the maximum amount of supported number series is reached. It is recommended to use two outgoing number series from a switching intersection. If a merge intersection is far away from the closest switching intersection, I recommend to use two incoming number series.

When the guideway layout is changed by adding or removing track sections, such as stations or intersections, only the connected number series is affected. For this reason, it might not be the best solution to divide the guideway into as few number series as possible if frequent changes are expected. In figure 8.2a, removing one of the stations would cause a change to all the intersections because the number of sleepers in an intersection is different than the number of sleepers in a normal track segment of the same length. Because of this, a new number series will be started at each of the tracks in splitting intersections, illustrated in figure 8.2b. To limit the amount of initial parameters, the system does not support transition between number series outside intersections.

Chapter 9
Control System Design

This section explains the general design principles of the control system. The first section describes the different units, how they interact and their main responsibilities. The second section describes how the guideway is divided into different control areas to form a distributed system and how the handover between different control areas are handled.

9.1 Using Embedded System Design

I have decided to use an embedded system design, which assings specific tasks to each unit in the control system. This makes the different units in the system less dependent on each other because they able to do their separate tasks withouth constant communication with each other. Further, this design principle leads to asynchronous communication between the units. By asynchronous, I mean that messages can arrive in any order at any given time, which in turn reduces the chance of deadlock situations if something unexpected happens.

The system consists of 4 main units where each unit has their own responsibilities. The responsibilities can be categorized in three different levels based on how much the decisions influence the rest of the system. The units and their corresponding responsibility level is illustrated in figure 9.1. The management layer takes all the strategic decisions in the system, the safety layer is responsible of the tactical decisions while the operational layer simply executes orders given from the safety layer. No units on the operational layer has influence on any other units than itself. No decisions made on the management layer has any immediate effect on the operation layer because all operations must be approved by the safety layer.

9.1.1 Distribution of Responsibility

The central is located on the top of the command chain. It has full control of all the other units in the system. The central manages the distribution of orders to the

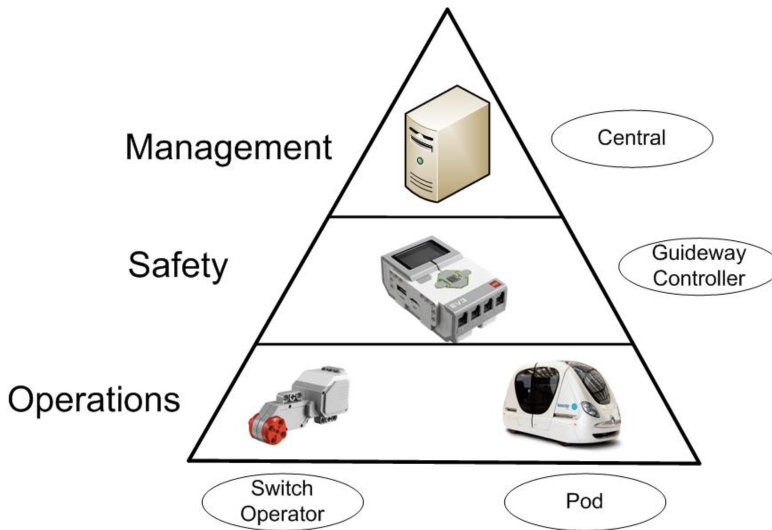


Figure 9.1: The different layers of responsibility in the system. The management layer makes no decisions that affect the lower layers directly. The safety layer needs to approve each and every action happening on the operation layer. All the physical movements happen on the operations layer. This layer will only follow orders from the safety layer.

pods and keeps track of the order status as well as the pods locations. Because it's the only unit knowing the location of all the pods, it is responsible of calculating the expected travel times along the different paths based on the traffic load. If there are any changes it will distribute updated routing tables to the Guideway Controllers.

The Guideway Controller has two main responsibilities. First and most importantly, it makes sure that the pods don't crash by keeping track of their position and authorizing only one pod to be at the same place at the same time. Second, it is responsible for switching the pods in the correct direction by sending commands to the switch operator. All of it's decisions affects the safety of the system. While the central decides the destination of the pod, the Guideway Controller decides when the pod can move and how far.

The Switch Operator receives commands from the Guideway Controllers stating which direction the tracks in a switching intersection should be pointed. When it receives a command from the Guideway Controller it will switch the position of the tracks if they are not already in the intended position.

The pods main responsibility is to move to the exact point it is authorized to by the Guideway Controller in an efficient way. To be able to do this, it must keep

track of it's position and speed in order to calculate the required braking distance to make sure it doesn't pass the point it's authorized to. In addition, it constantly reports it's speed and position to the central and guideway controllers. Finally, the pod manages the orders it is given by the central and reports back to the central every time an order status changes. The pod is not aware of other pods position, which control area it's in or which guideway controller it is communicating with. It only needs to be aware of its position and how many sleepers it is to the stop sleeper given in the last movement authority.

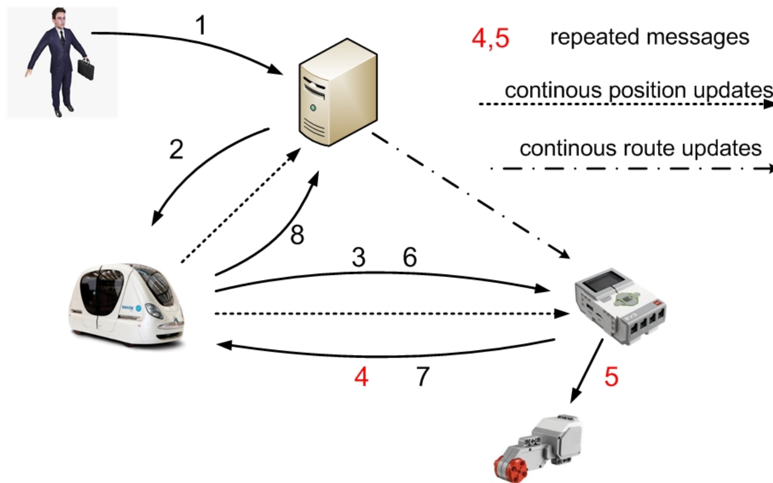


Figure 9.2: The most essential interactions from an order is generated until the pod is at the departure station. The dotted line from the pod to the central and guideway controller is location updates that are sent continuously. Route updates are sent from the central to the guideway controllers every time the travel times between stations change.

9.1.2 Essential Interactions

To get a better understanding of the functionality of each unit, figure 9.2 illustrates some of the most essential interactions between the different units in the system. Each interaction is explained below.

1. A customer places an order on a ticket machine or a smart phone. The order is sent to the central.

2. The central assigns the order to the closest pod

3. The pod informs the Guideway Controller of it's new destination

4. The Guideway Controller allows the pod to proceed to a given sleeper number via a movment authority. The given sleeper number can be far away from the pods current position or only a couple of cm. This depends on the position of other pods and the traffic load. The pod is repeatedly given movement authorities until it reaches it's destination station.

5. Every time the pod approaches a switching intersection, the Guideway Controller checks the routing table to find the fastest direction and commands the Switch Operator to switch the tracks in the correct direction.

6. The pod stops at the sleeper number given in the movement authority and reports standby to the Guideway Controller.

7. If the pod is standby at the given stations platform, the Guideway Controller notifies the pod that it has arrived it's destination station.

8. The pod sends an order update to the central.

9.2 Using Distributed Control

To make the system even more robust and scalable at the same time, the guideway is divided into separated control areas with one Guideway Controller in command of each of these areas. This makes it possible to expand the guideay withouth increasing the traffic or computational load on each Guideway Controller because the size of each existing control area will stay the same. Another fortunate effect of distributed control is that one inoperative guideway controller will only affect it's control area instead of the whole guideway.

9.2.1 Dividing the Guideway into Control Areas

A control area is defined as the area a guideway controller is allowed to issue movements authorities to pods. Control areas do not overlap, which means that two Guideway Controllers can not issue a movement authority to the same sleeper. The borders between control areas are always located directly in front of switching intersections and merging intersections on the side track. This means that a pod merging into an intersection from the straight track will stay in the same control area while it will change control area if it enters a merging intersection from the side track or a switching intersection. This also implies one switching intersection per control area. Figure 9.5 illustrates the LeMiP guideway split into control areas while figure 9.4 shows exactly where the borders are located.

At switching intersections, the control area starts one sleeper before the switching plate. This means that the previous guideway controller cannot authorize a pod to move it's front wheels into a switching intersection it doesn't control, which could

disturb a switching operation and potentially derail the pod. Since it's only the guideway controller in charge of the intersection that can allow the pod to enter it, the switching process will always be finished before the pod enters an intersection. At merging intersections, the control area starts at the red adjustment sleeper on the side track, just like the number series. This makes sure no pod can enter a merging intersection unless it has a clear path without any risk of crashing into a pod entering the intersection from the other track.

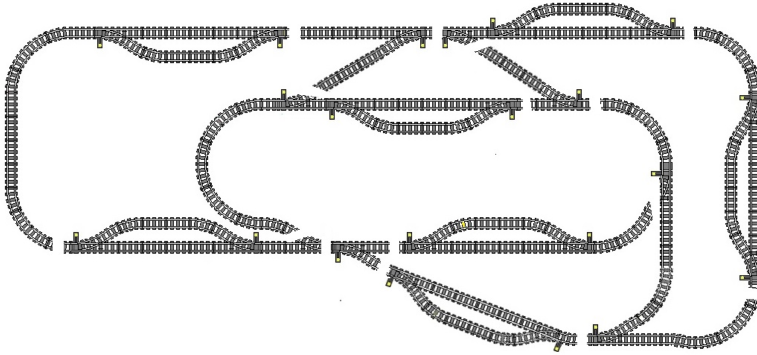


Figure 9.3: The guideway is split into 11 control areas, one per switching intersection. The borders between control areas are placed in front of switching intersections and merging intersections, but only on the side track.

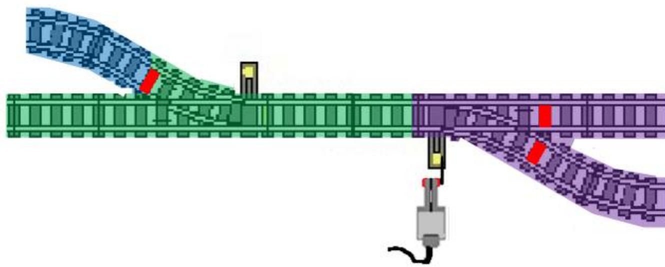


Figure 9.4: The different colors mark the different control areas. The red sleepers are adjustment sleepers, which indicates the transition between number series. At merging intersections, the control area starts at the adjustment sleeper on the side track. The straight track entering a merging intersections stays in the same control area. At switching intersections, the control area starts 1 sleeper before the switching plate.

9.2.2 Handover Procedure Between Control Areas

When a pod approaches a new control area, a handover procedure is initiated by the guideway controller currently in control of the pod. The handover process intends to make the transition between control areas efficient, but safe at the same time.

Because guideway controllers essentially only listens to the location updates from the pods in its own control area, it needs to be made aware of an incoming pod. When a pod is given a movement authority to the last sleeper in the current control area, the guideway controller sends a handover message to the next guideway controller. This message contains the pods unique id, which is needed to establish contact with the pod. If the guideway in the new control area directly in front of the pod is available, the new guideway controller allows the pod to enter the new control area. When the pods back end is clear of the the previous control area, the previous guideway controller disconnects from the pod.

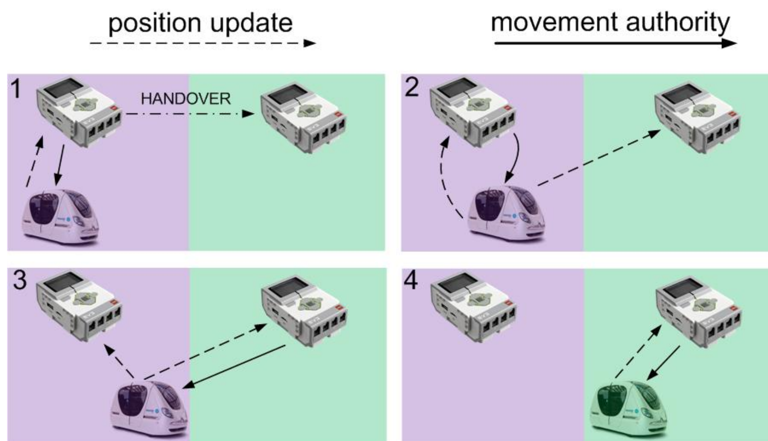


Figure 9.5: 1:The pod is given a movement authority to the last sleeper in the current control area and the guideway controller sends a handover message to the next guideway controller. 2: The next guideway controller connects to the pod and receives the location updates. 3:When the pod is closing in, it is issued a movement authority into the next control area. 4: When the pods back end is clear of the previous control area, the previous guideway controller disconnects from the pod

Chapter 10 Interactions

This chapter lists all the iteractions, i.e. the communication protocols, between the different units. I have grouped the different interactions into operational and administrative communication.

10.1 Operational Communication

I have defined operational communication to be the communication needed to move the pod safely along the guideway. The most crucial message is the movement authority, while the rest of the messages basically collects information needed to decide if a movement authority should be issued or not.

10.1.1 Movement Authority

The movement authority is issued by the Guideway Controller to the POD and notifies the pod that it's reservation has been expanded. A movement authority contains a stop sleeper and the maximum allowed speed level. When the pod receives a movement authority it will immediately proceed to the given stop sleeper unless it is located at a station platform with it's doors open. In this case, it will execute the movement authority as soon as the doors are closed.

10.1.2 Switch Orders and Confirmations

Every time a pod requires switching, the guideway controller sends a switching order containing the direction to the switch operator. When the switch operator receives the order, it will switch the direction if needed and confirm the order. The confirmation is crucial to make sure the tracks are in position before the pod is given a movement authority by the guideway controller.

10.1.3 Location Update

Location updates are sent from the pods to the central and listening guideway controllers. To not congest the network with location updates, only each fifth sleeper is reported. In addition to the sleeper number, the updates also contain the current speed of the pod. When a pod reached the stop sleeper given in the movement authority, it stops and sends a location update with the speed set to 0.

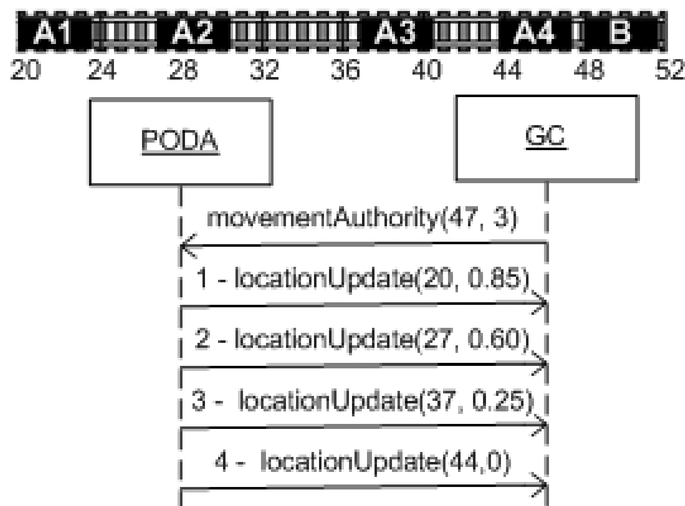


Figure 10.1: Illustration of the communication between PODA and a guideway controller(GC). PODA is only given movement authority to sleeper 47 because PODB is standby at sleeper 48. Notice that location updates states the location of the pods back end while movement authorities states the furthest allowed location of the pods front end.

10.1.4 Arrival Notice

When a POD reports standby next to the destination station platform, the current guideway controller sends an arrival notice to inform the pod that it has reached it's destination.

10.1.5 Handover

When a pod is given movement authority to the last sleeper in the current control area, the current guideway controller broadcasts a handover message to all the other guideway controllers. The handover message contains the pods unique id and the entrance sleeper to the next control area. The guideway controller in charge of the next control area will recognize the entrance sleeper and initiate contact with the pod in question.

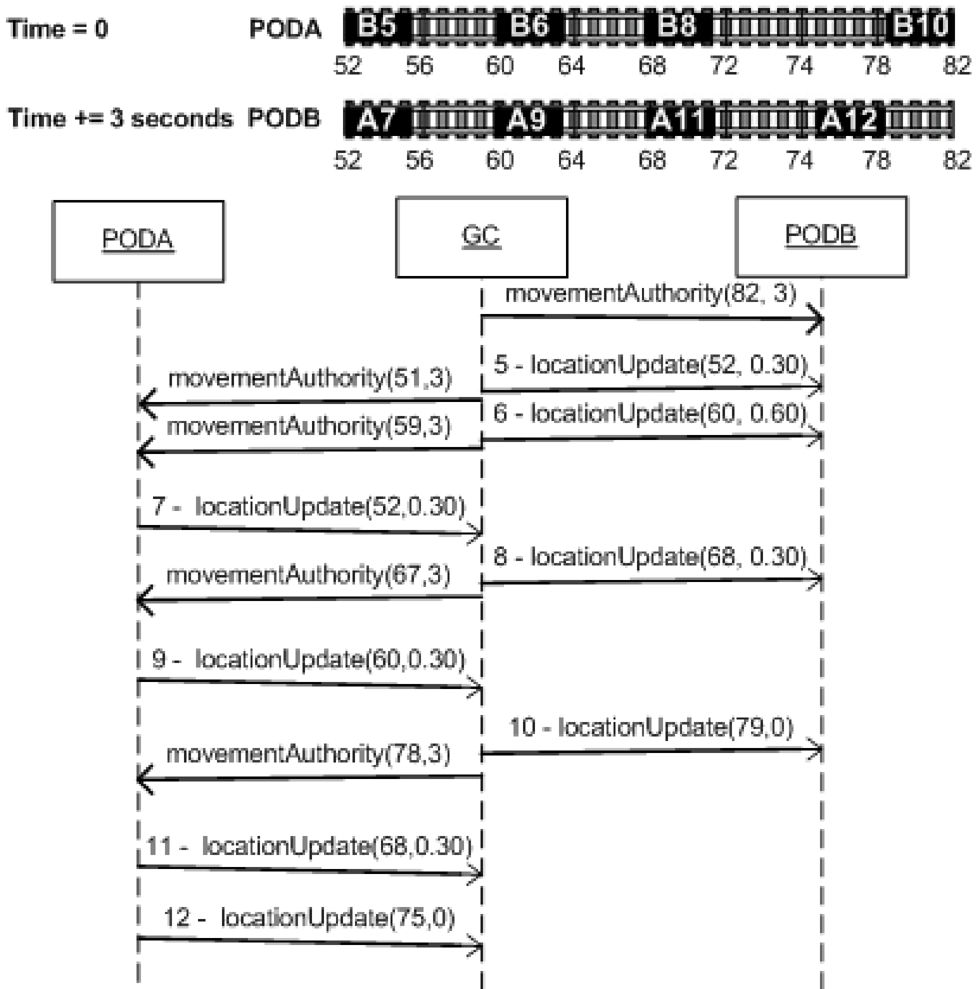


Figure 10.2: Illustration of the communication between PODA, PODB and a guideway controller(GC). The two tracks illustrates respectively PODA and PODB in the same area at different times. PODB is given movement authority to sleeper 82. Every time PODB sends a location update, PODA is given movement authority to the sleeper prior to PODBs last reported location.

10.1.6 Destination Request and Responce

When the POD is approaching a splitting intersection, the associated Guideway Controller asks the POD for it's destination in order to be able to switch the POD in the right direction. A POD is not issued a movement authority into the intersection before the Guideway Controller knows it's destination and the tracks are switched in the correct direction.

10.1.7 Adjustment Sleeper Information and Confirmation

Before a pod enters a new control area, the information about every adjustment sleeper it will pass in the control area is sent to the pod from the guideway controller in charge of the new control area. Each adjustment sleeper is described by an old number and a new number. The old number is the number it would have in the last number series, i.e the last sleeper number in the previous number series plus one. The new number is the first sleeper number in the new number series. The adjustment sleepers are listed in the same order as they will be passed by the pod. Because there are two optional paths through the control area if the pod enters via a splitting intersection, the guideway controller needs to know the pods destination before it can send the adjustment sleeper information.

Before a movement authority can be issued, the pod has to send a confirmation message to the guideway controller. The confirmation message is crucial because if the pod receives a movement authority before the adjustment sleeper information, there is a change that it will calculate the braking distance based on wrong information.

10.2 Administrative Communication

Administrative messages are a higher level of messages that is invisible from the operational layer or not needed to move the pod along the guideway. The administrative messages are needed to keep the system optimized, but the operational layer are not dependent on these messages.

10.2.1 New Order, Cancel Order and Order Status

An order is usually generated by a customer wanting to be transported from one station to another. The order always contains an unique order id, departure station and arrival station and is sent to a pod from the central. The central can also generate orders when it wants to redirect a pod. There are several reasons why the central would want to do this. If pods that deliver customers at a station isn't assigned new orders, the station will eventually fill up. To prevent this, the Central relocates the pod by issuing a new order with the departure station set to a station with sufficient capacity. Such orders will have the arrival station set to null. Another reason might be to line up pods at stations that are expected to be busy at rush hour.

The central can cancel an order at any time as long as it's currently not being processed. In the opposite case, the central must make sure that the pod has received a new order it can replace with the cancelled one. If not, the pod will refuse to cancel the order because it always needs a destination station unless it's already at

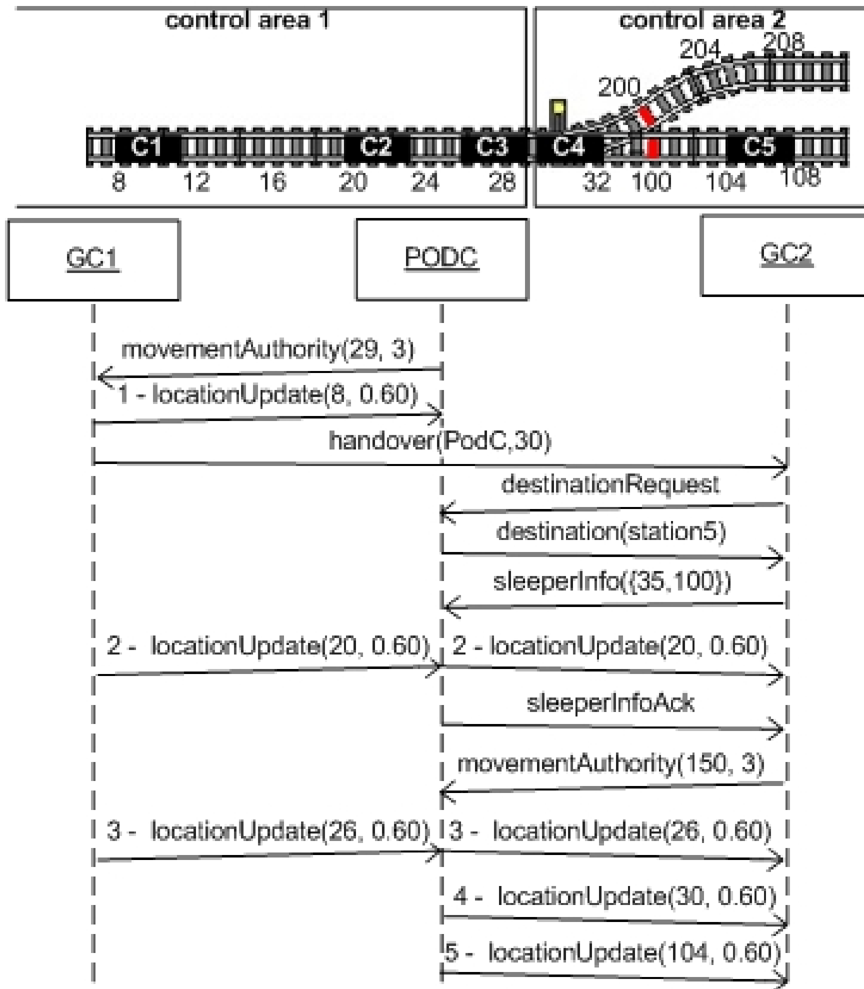


Figure 10.3: Illustration of the handover procedure. When guideway controller 1(GC1) has issued movement authority to the last sleeper in its control area, it broadcasts a handover containing the pods id and first sleeper in the next control area, called the border sleeper. GC2 initiates communication and immediately receives the retained destination message. When the GC knows which direction to switch the pod, it sends the adjustment sleeper information. When the pod confirms the reception of the adjustment sleeper information, the pod is given movement authority into control area 2.

a station. This makes sure that a pod located on the transit track always has an active order and a destination station.

The status report contains the order number and current status. From the pods

perspective the order status is set to "assigned" at once it's received. When the order is processed, the status changes to ”enRouteDepartureStation'. When the pod arrives a station, it checks if the destination station is set. If so, it means that a customer needs to be picked up and the pod changes the order status to ”enRouteDestinationStation”. If not, it means that the pod has been reallocated by the Central and the pod changes the order status to ”finished”. If the order is cancelled, the pod confirms by sending the order status ”cancelled” back to the Central. The life cycle of an order is illustrated in figure figure 10.4.

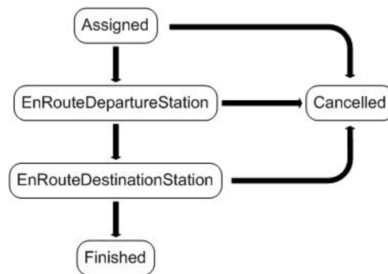


Figure 10.4: Order life cycle

10.2.2 Route Updates

Initially, the routing tables based on the distances between stations are sent to each guideway controller. In addition, the central can update routes based on the actual travel time by keeping track of the pods. When a change in travel time causes the fastest routes to change, the central can send an update to the affected guideway controllers. This will make the guidewaw controllers able to switch the pods based on live traffic info.

10.3 Initial Communication

Initial communication is needed at every system startup and to move every single pod from the start track into the guideway. Withouth the initial communication the system will never become operational. The initial communication is sent from the central to the pods and guideway controllers and contains dynamic information, such as pod id, guideway layout and station names. Distributing this information at startup in stead of including it in each units code reduces the work significantly when the information needs to be changed. In the opposite case, changes would have to be distributed manually to each unit.

10.3.1 Registration, Control Area, Station Name and Pod Id

When the guideway controllers and pods has started their programs, they register to the central. guideway controllers are initially aware of their id and includes it in the registration message. The central looks up the id in a table and sends the control area information to the guideway controller. This information consists of all the intersections located in the control area. If the control area contains a station, the name of the station is also sent from the central to the guideway controller.

To avoid having one program for each pod, the pods are not aware of their id at start-up. The central will assign the pods ids consecutively, which means that the first pod will get id POD001, the second id will be POD002, etc. To keep track of the pods visually during system operation, they must be marked physically with their id and started up in the correct order.

10.3.2 Movement Authority and Handover

When a pod has received it's id, the central issues a movement authority to the end of the start track before it broadcasts a handover message to all the guideway controllers. The following procedure is exactly like any other handover. The only difference is that the central acts like a guideway controller. When the back end of the pod is clear of the start track, the next pod is processed the same way. It is important to remember that the central processes the pods in the same order as the they were registered. This means that the pods would crash into each other if the registration order and line up order on the start track isn't the same.

Chapter 11

The Guideway Controllers Internal Functionality

This chapter explains the internal functionality of the guideway controller. To be able to understand the functionality the first sections explains the MQTT topic hierarchy and how the control area is divided into sections and reservations. The next sections explains how the reactive blocks are organized in a hierarchy with different levels. The levels are explained from the top to the bottom where each level, except the lowest one, is explained in a main section. There is a total of 5 levels. Only one block on each level will contain lower level blocks. This block will be explained in a new section while the blocks containing no further levels are explained in subsections. As a result, the guideway controller, control unit, guideway manager and movement authority manager are explained in the main sections.

11.1 MQTT Topic Hierarchy

To decrease the amount of messages being received by the different units, the topic hierarchy is organized to make sure that no units receive messages not intended for them.

There are five main branches of topics.
LEMIP/CENTRAL/message category
LEMIP/GC/id, BROADCAST/message category
LEMIP/SO/id, BROADCAST/
LEMIP/TO_POD/id, BROADCAST/message category
LEMIP/FROM_POD/id/message category

Levels containing non capitalized words is variables. For example the id can be ”POD001” or ”GC005”. When a level contains ”id,BROADCAST” it means that the level can either contain a unique id or ”BROADCAST”, which means that it is intended for everyone subscribing to the topic branch. The ”message

category" variable classifies what the content of the message is, for instance it can be "HANDOVER", "LOCATION_UPDATE", "MOVEMENT_ATHORITY", etc.

11.2 Dividing the Control Area Into Track Sections

Each control area is divided in sections. The length of each section depends on how the control area is designed. However, general rules decide how the sections are divided based on the location of the intersections. One section cannot span longer than one number series, but one number series can span over many sections.

In switching intersections, there is a small section between the start of the control area and the first adjustment sleeper. This is later referred to as the switch section. Even if there is no adjustment sleeper on the straight track exiting the switching intersection, the switch section will end at the same sleeper id as the last sleeper in the number series on the side track. The sectioning of switching intersections is illustrated in figure 11.1. The rest of the control area is divided in one additional section per merging intersection as illustrated in figure 11.2.

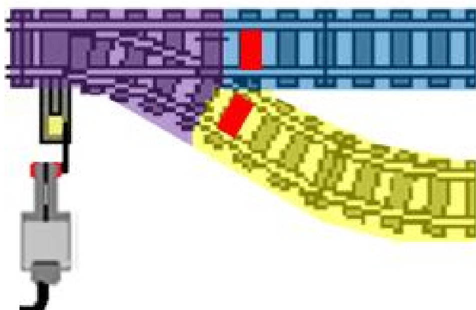


Figure 11.1: The division of sections at switching intersections. An incoming pod needs to pass the purple section(switch section) regardless of the direction. If a pod has entered the blue section, the next pod can enter the yellow section through the purple section as long as the first pods back end is clear of the purple section and vice versa.

When a pod approaches a new section, it is registered in the section in a First In, First Out(FIFO) manner. When the back end is clear of a section, the pod is unregistered. This procedure makes it easy to find the pods positions relative to each other. It also serves as a the order of priority when two pods are approaching the same merging intersections from different tracks. By changing the requirements for being registered in a section, it is in fact the priority policy that is changed. Currently, the pod needs to be closer than 20 sleepers in order to be registered in the next section.

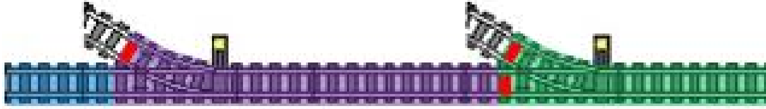


Figure 11.2: The division of sections at merging intersections. The sections are divided the same way as number series, but in contrast to number series, there must be one new section per merging intersection. While the blue and purple sections are in the same number series, the green is in another divided by an adjustment sleeper.

11.3 Reservations

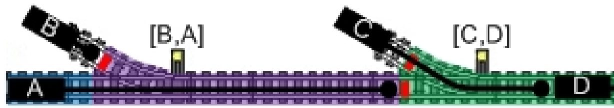
To make sure two pods cannot cross each others paths and crash, they are assigned reservations prior to sending a movement authority. Only one pod can reserve the same sleeper at the same time and one pod cannot have two discontinuous reservations. The stop sleeper in a movement authority is always the last reserved sleeper. This procedure fulfills the ”brick-wall stop” criterion.

There are two scenarios where a pods reservation changes. When a pod reports a new location, the first sleeper in the reservation is moved to the new location. Before a pod is given a movement authority, the last sleeper in the reservation is moved to the last available sleeper. The last available sleeper in a section is one sleeper prior to the next pod or the last sleeper in the section if there are no pods ahead in the same section. A location update always contains the location of the pods back end, not the location of the color sensor. This imply that a pod cannot have a reservation shorter than it's own length at any given time. To sum up, the number of reserved sleepers increase every time a movement authority is issued and decrease every time the pod reports a new location.

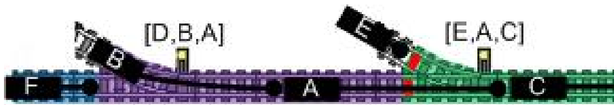
Because the guideway controllers aren't aware of pods locations prior to receiving a handover, pods aren't reserved sleepers in a section unless it's registered in it. Since a pod needs to be closer to a section than 20 sleepers to be registered in it, a pod isn't assigned any reservations in a section unless it is closer than this distance. This makes sure two approaching pods gets priority in merging intersections based on their distance to the intersection. Other rules can be applied, for example giving a pod with passengers priority over an empty pod. However, a stalled pod causes a chain reaction to all succeeding pods. Because the guideway controller is only aware of the first pod in line at the side track, it's impossible to predict the consequences with the current distributed system design. For this reason, only one pod on the same track is merged at the same time if there are pods on both incoming tracks.

Let's look at an example illustrating a likely situation. In figure 11.3a, pod C was handed over from the previous guideway controller after pod A was issued

movement authority to the end of the purple section. If no sections or registration rules existed, pod A would be given movement authority into the green section because the guideway controller wasn't aware of pod C yet. In this case, pod C would have to wait for pod A to pass. This is bad capacity utilization because there is more than enough space and time for pod C to merge between pod D and pod A as illustrated.



(a) Pod A reserved the sleepers in the purple section before pod B because it arrived the merging intersection first. Similarly, Pod C reserved the available sleepers prior to Pod D in the green section before Pod A.



(b) This is the same area as infigure 11.3a, but some moments later. Pod D has moved, which causes a chain reaction behind it.

Figure 11.3: figure (a) and (b) illustrates how pods are prioritized when assigned reservations. Each color represents a section. The guideway without color is in a different control area. The registration order, included inside the square brackets, is the order the pods are given authority to enter a new section. The pods reservations are illustrated with a black line. The last sleeper in each reservation is marked by a black dot.

11.4 The Guideway Controller Block

The top layer of the guideway controller consists of 3 blocks, illustrated in figure 11.4. The blocks on the sides are MQTT communication blocks, which are included in the Reactive Blocks libraries. The one to the left is the ”MQTT subscribe” block, which receives messages while the one to the right is the ”MQTT publish” block, which sends messages. The control block, which contains the guideway controllers functionality is located in the middle. Messages to and from the control block is marked by green flows. To administer which topics the MQTT subscribe block subscribes to, the two purple flows respectively adds and removes topics.

11.4.1 MQTT Subscribe

Initially I used the ”Robust MQTT” block, which is included in one of the Reactive Blocks libraries to both send and receive messages. Because the list of subscriber topics cannot be altered after the block is initiated, each guideway controller subscribed to

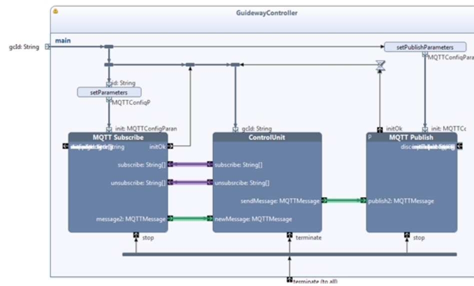


Figure 11.4: The control unit is the block in the middle. Located on the left side is the ”MQTT subscribe” block, which receives messages while ”MQTT publish” block located on the right sends messages. The green flows are messages sent to and from the control unit. The purple flows are lists of topics that should be subscribed or unsubscribed.

”LEMIP/FROM_POD/#”, which means that each guideway controller receives all the messages from all the pods.

Subscribing to messages of no interest is not an elegant solution because most of the received messages will be discarded. This causes an increase in network load because messages in practical terms will be broadcasted when it only needs to be sent to one specific guideway controller. Further, filtering all the interesting messages from all the uninteresting ones will consume more computation power than needed because this is in fact the brokers job.

To make the MQTT client able to alter the subscribed topics every time a pod is handed over, I decided to use the ”MQTT Subscribe” block for receiving messages and the ”MQTT Publish” block for sending messages. The MQTT Subscribe will start a new instance of a MQTT client for each subscribed topic. When the topic is unsubscribed, the associated client instance is terminated.

Initially, the MQTT subscribe is started by sending parameters to the init pin on the top of the block. These parameters contains the address for the MQTT broker and the fixed topics, which is LEMIP/GC/id/# and LEMIP/GC/BROADCAST/#.

11.4.2 MQTT Publish

Sending messages is much simpler because the topic of each published message is specified inside the control unit. The only parameters needed initially is the broker address, which is sent to the init pin on top of the MQTT publish block.

11.5 The Control Unit Block

The control unit contains 5 blocks as illustrated in figure 11.5. The block is initiated by the init pin located under the block, which contains the guideway controllers id. The id variable is needed by the classify message block and the format message block. The termination pin is triggered to stop the program. In addition, the block has 2 more input pins and 2 more output pins. These are the ones marked by colors in the higher level "guideway controller" block.

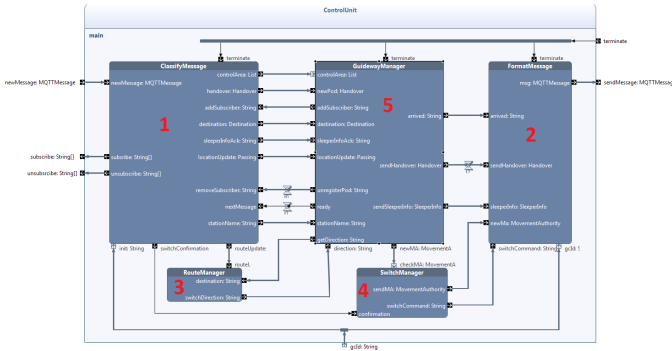


Figure 11.5: The control unit contains 5 blocks: classify message(1), format message(2), route manager(3), switch manager(4) and guideway manager(5).

11.5.1 Classify Message

All messages send to the guideway controller has to go via this block to be buffered, filtered and classified before it can be distributed to the correct block for processing via 1 of 7 output pins. If the message topic isn't recognized, the message is dropped before it reaches the classification operation.

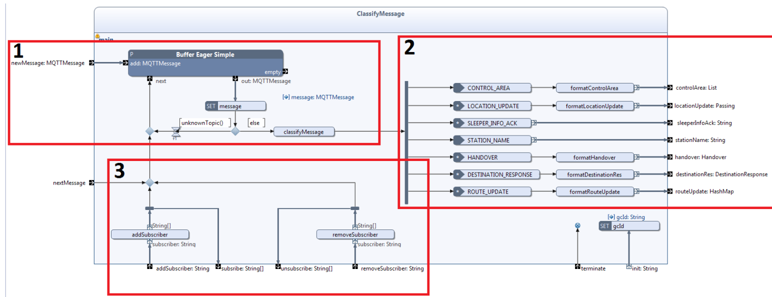


Figure 11.6: The whole "classify message" block. The block is divided in three by red rectangles. Each of the rectangles are illustrated separately below.

Despite its name, Message Queue Telemetry Transport does not support message queueing. This means that when a message is published to a topic, the broker will immediately push it to all subscribers. Because of this, a new message can arrive before the guideway controller is finished processing the previous one. In some cases, this will cancel the the processing of the previous message, which could cause a deadlock in the system.

To prevent a newly arrived message to disturb the processing of the previous one, incoming messages are stored in a buffer. Every time the guideway controller is finished with a message, it pulls the next message from the buffer by triggering the ”next” pin on the buffer block. This will either be done if a message is dropped by the topic filter or when the guideway controller has finished the processing of a message. In the latter case, the ”next message” input pin is activated from the ”guideway controller” block. If the buffer is empty, it will wait for a new message. The buffering is solved by implementing the ”Buffer Eager Simple” block, which is included in the Reactive Blocks standard libraries.

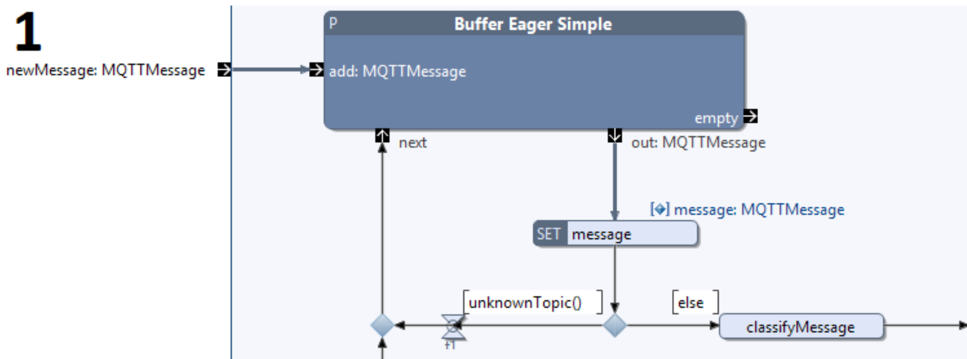


Figure 11.7: Messages are buffered to process them one by one. If a message from a newly unsubscribed topic is located in the buffer, it needs to be filtered out before it reaches the ”classify message” operation.

Messages are buffered in order to process them one by one. As a result, messages from unsubscribed topics might be inside the buffer. To filter out these messages, the topic of each message must be verified before the message is processed. If the topic cannot be recognized, the message is dropped and the next message is pulled from the buffer by triggering the ”next” pin on the buffer block.

If the topic is recognized, the message is classified based on the topics third level. Because the payload of each message is a string, in most cases, it needs to be formatted to match the object of the belonging output pin. When formatting is needed, the different variables included in the message string is arranged in a

specific order and separated by commas by the sender. After formatting, the message is distributed to the correct block via the output pin. The ”sleeper info ack” and ”station name” message doesn’t need formatting because the message only contains one string variable.

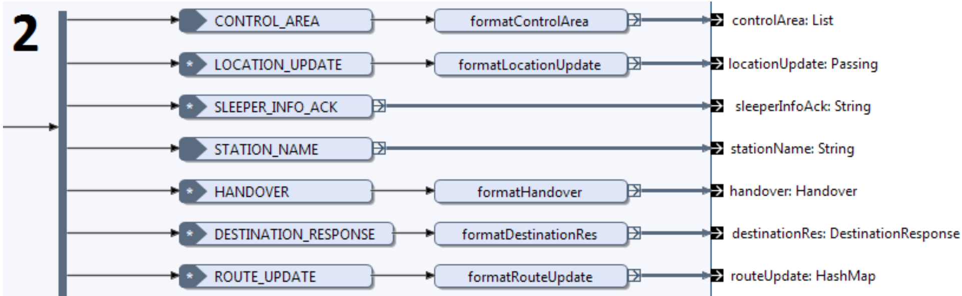


Figure 11.8: Messages are formatted from a comma separated string into an object and sent to a separated output pin. In some cases, formatting is not needed because the output pin expects a string.

When a pod is added or removed, the topic list used by the filtering mechanism needs to be updated as well as the list sent to the ”MQTT subscribe” block. This is also handled inside the ”classify message” block.

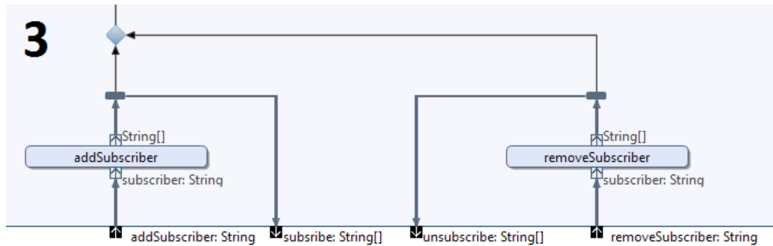


Figure 11.9: When a pod is handed over, it is either added or removed from the topic list depending if it’s arriving or leaving the control area. This requires the list used by the topic filter and the ”MQTT subscriber” block to be updated continuously

11.5.2 Format Message

All messages sent from the guideway controller must go via this block to be formatted into a valid MQTT message. The message is formatted and assigned a topic based on the input pin. In stead of distributing messages to different pins, like in the classify message block, the format message block gather the different messages into one pin.

11.5.3 Route Manager

The route manager block contains the routing table that contains all the stations in the system. When the guideway manager asks which direction to switch a pod, the route manager looks up the destination station in the routing table and returns either ”straight” or ”side”. To keep the routes up to date, the route manager can receive route updates from the central. If the route manager can’t find the requested station in it’s routing table, it will return ”straight” by default.

11.5.4 Switch Manager

To make sure a movement authority isn’t sent to a pod in front of a switching section before the tracks are in the correct position, each movement authority needs to be approved by the switching manager. The movement authority object contains a flag that is set to true by the movement authority manager if the pod is entering a switching intersection. In this case, the switch manager will hold back the movement authority by storing it before sending the switching order to the switch operator. When it receives a confirmation that the switch operation is finished, the movement authority is passed on to the pod via the format message block.

If other movement authorities arrives while the switch manager is awaiting a confirmation from the switch operator, it will be relayed immediately without a check. This is safe because there is only one switch intersection per guideway controller, which means that only pod movement authority with the switch flag set to true can exist at the same time.

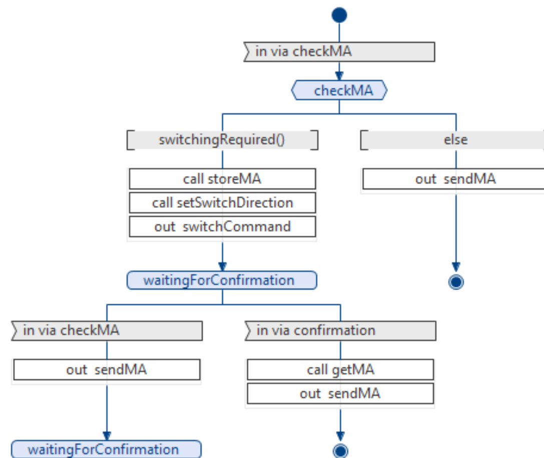


Figure 11.10: State Machine Diagram of the switch manager.

11.6 The Guideway Manager Block

The guideway manager block contains the main functionality of the guideway controller. It is responsible for creating the control area, keeping track of the pods and issue movement authorities. To be able to do this, it contains 3 lower level blocks and a lot of operations. This block is the most chaotic one. To make it easier to understand how the different operations are connected, they are grouped in colors. Each color, except yellow and black, represents a specific chain on operations that are activated based on the pods position. But let's start by taking a look inside the two first blocks.

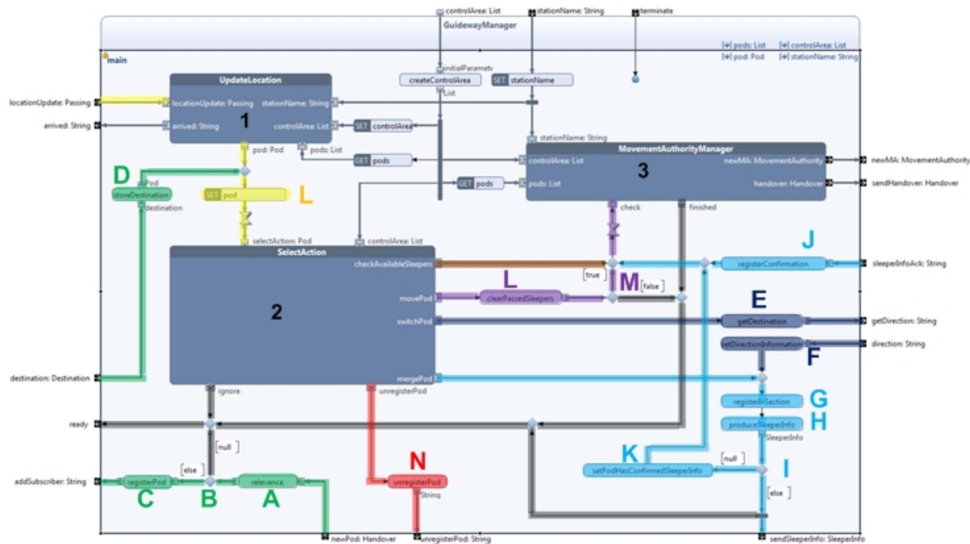


Figure 11.11: The guideway manager block contains 3 lower level blocks: update location(1), select action(2) and movement authority manager(3). In addition, there is a lot of operations(A-N). To make it easier to see the relations between them, associated operations and the flows between them are grouped in colors.

11.6.1 Update Location(1)

This block is responsible of updating the pods location and check if the pod has arrived at the station platform. To be able to do this, it needs to initially receive the reference to the control area, list of pods and the station name via the corresponding input pins. In addition, the "location update" input pin is triggered every time a pod reports it's location, which contains the speed and location of the pod. The output pin "pod" contains the instance of the pod who reported it's location. The "arrived" output pin is only triggered if the pod has stopped at its destination stations platform. The procedure is described in detail below.

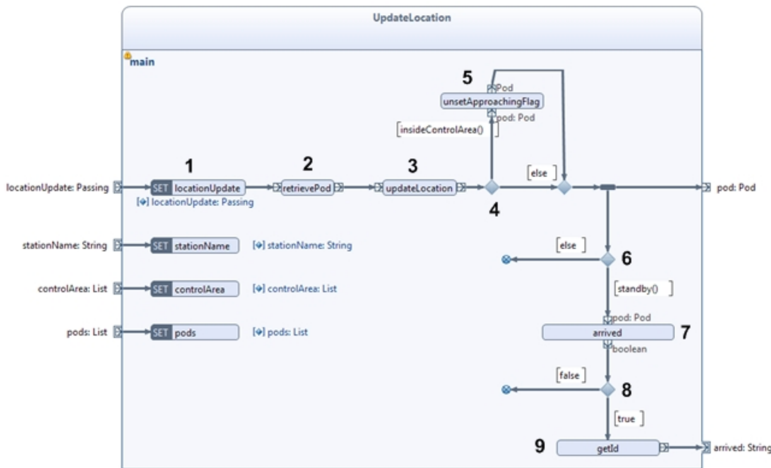


Figure 11.12: The update location block contains an activity diagram describing the update location procedure.

When the block receives a location update, the following steps will be executed.

1. The location update is stored to make it available by the following operations.
2. The pod is retrieved based on the given pod id in the location update.
3. The location of the pod is updated
4. The decision node will direct the flow upwards if the pods reported position is inside the control area. If not, it will direct to flow to the right.
5. If the reported position is inside the control area, the approaching control area flag, which is set to true when a pod is registered, is set to false. This flag makes it possible to distinguish between pods approaching the control area and pods that has left.
6. regardless of the decision in step 4, the flow will be directed both to the next decision node and to the "pod" output pin. The decision node will send the flow to the left and into the flow final if the pod is not standby. If it is, the flow will be directed downwards.
7. If the pod is at standby, this operation will return true if the pod is located at the station platform.
8. If the pod is at standby at the station platform, the flow will be directed downwards. Otherwise, the flow will be directed to the left and into the flow final.
9. If the pod has arrived its destination station, the id will be retrieved and sent via the "passing" pin to the "format message" block where it is formatted into a MQTT message.

11.6.2 Select Action(2)

When a location update or destination message is received, the select action block is activated. Based on the pods last reported location, the block will select one of 6 output pins which determines which of the colored flows that will be activated and in turn which operations that will be carried out. The pods can either be merged, switched, unregistered or moved. In addition, it can be checked for available sleepers if it is closing in on a control area or ignored if it's too far away. To be able to do this, the block initially needs to receive the control area via the corresponding input pin. The select action procedure is illustrated in figure 11.13 and explained step by step below.

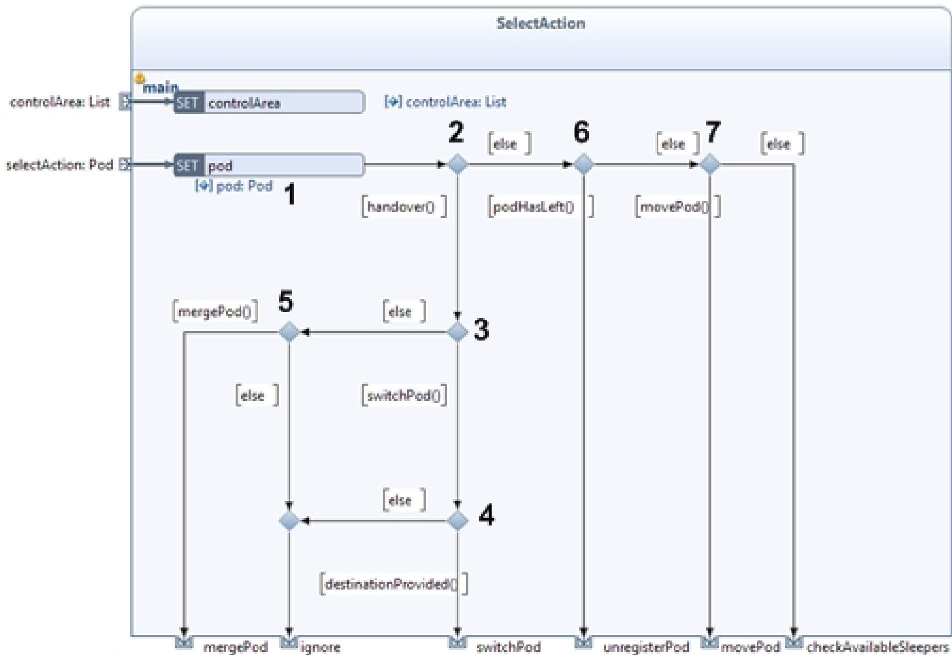


Figure 11.13: The inside of the select action block, which determines which of the flows that should be triggered in the "guideway manager" block on the above level. The procedure is described by an activity diagram.

When the block is triggered via the select action pin, the following steps will be executed:

1. Initially, the pod instance is stored to be available for the following operations
2. When a pod is registered, the handover flag is set to true. This indicates that the pod is approaching the control area, but is still in the handover phase, which means that it has not been registered for switching or merging yet because it's

too far away from the intersection. If this is the case, the flow will be directed downwards.

3. If the handover flag is set to true, the block will first check if the pod is approaching the switching intersection and if it's close enough. If these two conditions are met, the flow will be directed downwards. In the opposite case, the flow will be directed to the left.
4. If the pod is approaching a switching intersection and is close enough to be switched, it's crucial that it has provided it's destination. Otherwise, the guideway controller won't know which direction to switch it and will take no further action by directing the flow to the ignore pin, which will trigger the black flow in the higher level "guideway manager" block.
5. If the conditions in step 3 aren't met, the decision node will check if the pod will enter the control area via a merging intersection. If the pod is close enough, the flow will be directed to the "merge" pin, which will trigger the light blue flow. If not, the flow will be directed downwards to the "ignore" pin, which will trigger the black flow.
6. If the pods handover flag is set to false, the next step is to figure out if the pod has left the control area. This is done by checking if the pods last reported location is outside the control area. If it is, the flow will be directed downwards to the "unregister pod" output pin, which will trigger the red flow. If not, the flow is directed to the right.
7. If the pod hasn't left, there are two options left. Either it is inside the control area or less than 20 sleepers ahead of it. In the first case, the first sleeper in the pods reservation needs to be moved to the pods current location and the flow is directed to the "move pod" output pin, which triggers the purple flow. In the second case, the pod either has no reservation or it's back end hasn't entered the reserved area yet. In both cases, the guideway controller directs the flow to the "check available sleepers" pin, which triggers the brown flow.

11.6.3 The Yellow and Black Flow

The yellow flow is triggered every time the guideway controller receives a position update from a pod. After the location update is processed by the "update location block", the returned pod instance is stored in step L before the yellow flow ends up in the "select action" block(2), which determines what to do next based on the pods position.

The black flow is activated when a received message is processed and a new one needs to be pulled from the buffer in the "classify message" block. A message is considered to be processed when there are no further operations along the current flow before it leaves the block. Because the black, green and red flows are merged

inside the ”classify message” block, it is not necessary to activate the black flow when the green or red flow leaves the guideway manager block.

11.6.4 Registering an Approaching Pod(green flow)

The green flow is triggered when the guideway controller receives a handover. If the entrance sleeper given in the handover is located in the control area, the pod will be registered and added to the subscriber list. If not, the handover message will be dropped and the next message will be pulled from the buffer in the ”classify message” block. The procedure is explained in detail below.

A. The first step is to determine whether the entrance sleeper given in the handover is located in the guideway controllers control area or not. If not, the handover message is discarded and null is returned. In the opposite case, the handover will be passed on to the next step.
B. The decision node direct the flow upwards if the flow contains a null, which will trigger the black flow. If not, it means that the handover is intended for the guideway controller, and the flow will continue to the left.
C. If the handover is intended for the guideway controller, this operation will register the pod by creating a pod instance, assign it the id given in the handover and add it to the list of registered pods. Finally, it will return the id of the pod to be registered in the topic list inside the classify message block, which in turn will sent the topic to the MQTT subscribe block.
D. When the guideway controller has successfully subscribed to the pods messages, the broker will immediately publish the pods destination directly. This is possible because the destination messages are retained, which means that the last destination message for all pods is stored by the broker. Operation D will store the destination and return the pod instance before the green flow merges into the yellow flow, stores the pods instance and activates the ”select action” block.

11.6.5 Switch an Approaching Pod(dark blue flow)

If the pod is entering the control area via the switching intersection, is close enough and has provided it’s destination, the ”select action” block will trigger the ”switch pod” pin. The procedure is similar for pods approaching switching and merging intersections. However, to determine which direction to switch a pod approaching the switching intersection, the ”route manager” block must provide the fastest direction to the destination station.

E. This operation retrieves the pods destination station before a direction request is sent to the route manager block via the get direction pin. The ”route manager”

block will immediately return either ”straight” or ”side” via the ”direction” pin.

F. The switch direction returned from the ”route manager” block is stored.

11.6.6 Merge an Approaching Pod(light blue flow)

If the pod is entering the control area via a merging intersection and is close enough, the select action block will trigger the merge pod pin. The procedure will be the same as when a pod is being switched, but without the first two steps(E and F), because there is only one possible path through the control area when a pod is merging.

G. When the pods meet the conditions for either switching or merging, it will be registered in the section it will enter the control area via.

H. Based on the path the pod will take through the control area, the list containing the adjustment sleeper information is produced. If the pod won’t pass any adjustment sleepers in the control area, null will be returned.

I. The flow will be directed to the left if null is returned from operation H. If not, the list of adjustment sleepers is sent to the format message block via the send sleeper info pin.

J. When a pod is registered, the confirmation flag will be set to false. This flag needs to be set to true before a pod can reserve sleepers in the control area. When the pod receives the adjustment sleeper info, it will return a confirmation. This will activate the register confirmation operation, which sets the confirmation flag to true. Finally, the flow is sent to the ”check” pin in the ”movement authority manager” block.

K. If the pod won’t pass any adjustment sleepers in the control area, the confirmation flag is immediately set to true before the flow is sent to the ”check” pin in the ”movement authority manager” block.

11.6.7 Check Available Sleepers(brown flow)

If the pod has been registered in the first section, i.e. either been switched or merged, but with the back end still outside the control area, the ”select action” block will trigger the ”check available sleepers” pin. This will immediately trigger the ”check” pin in the ”movement authority manager” block.

11.6.8 Move Pod(purple flow)

When the pods back end is located in the control area, a location update requires the first sleeper in the pods reservation to be moved to the given sleeper. This allows succeeding pods to reserve the newly freed sleepers. However, before the reservation is altered, the location update must be validated. To account for location updates arriving in the incorrect order, the update will be discarded if the given position is

outside the reservation area. This makes sure that reservation cannot be extended backwards, which can cause chaos if another pod has already reserved these sleepers. Moving a pod require two operations described below.

L. If a location update is discarded, operation L will return false. If the updated location is inside the reservation area, the reservation will be updated according to the new location and true is returned.

M. If operation L return false, the decision node will direct the flow to the right where it will merge into the black flow. In the opposite case, the flow is directed upwards and into the ”movement authority manager” block via the ”check” pin.

11.6.9 Unregister Pod(red flow)

N. When the back end of a pod is clear of the control area, the ”select action” block will trigger the ”unregister pod” pin. This will activate the unregister pod operation, which deletes all the pods reservations and registrations in the control area. The operation will return the pods id, which is sent to the ”classify message” block and removed from the topic list. Finally, the ”MQTT subscribe” block associated with the pods id will be terminated and no further messages will be received from this pod until it has travelled around the guideway and is approaching the control area again.

11.7 The Movement Authority Manager Block

The ”movement authority manager” block is responsible for checking if any pods can extend their reservations and be issued a new movement authority. In addition, it will generate a handover when a pod is given movement authority to the last sleeper in the control area. The block contains two lower level blocks and operations that can be categorized into 4 different flows illustrated in figure 11.14. The yellow flow is a loop, which checks one pod in each iteration until all the pods registered in the control area is checked, which triggers the black flow. If the green flow is triggered, it means that the pod being checked in the current iteration has extended its reservation and a movement authority is issued to make the pod aware of this. If the blue flow passes both decision nodes, a handover is generated and broadcasted to all the other guideay controllers. To be able to do this, the ”movement authority manager” block needs access to the control area and list of registered pods, which is initially stored when the corresponding input pins are triggered.

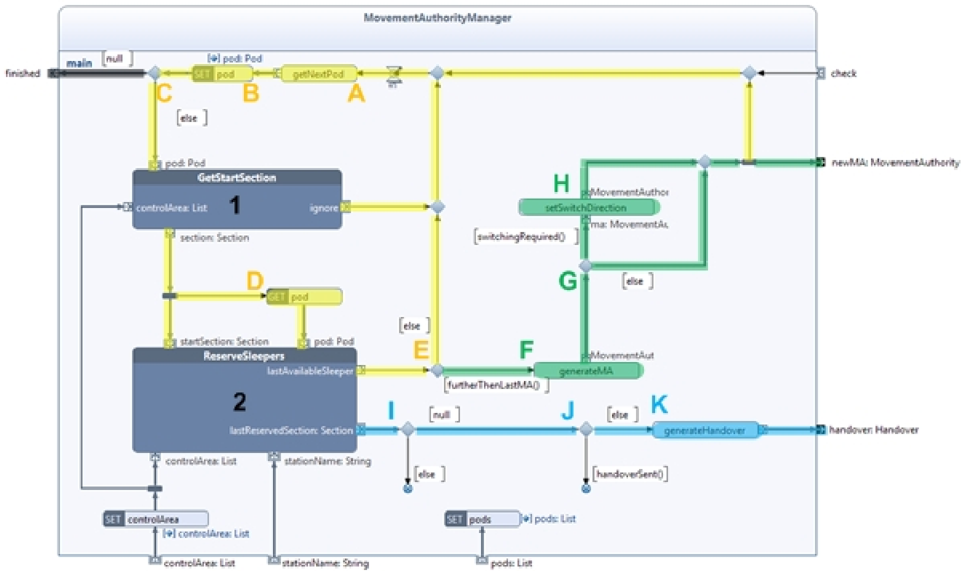


Figure 11.14: The movement authority manager block contains 2 lower level blocks and operations organized in 3 different groups based on the color.

11.7.1 The Loop(yellow flow)

When the "check" pin is triggered, the loop will iterate one time for each registered pod. The number of steps in each iteration depends on the returned value from the two lower level blocks. However, no reservations will be changed, no movement authorities will be issued and no handovers will be sent if the green flow isn't triggered. There are 2 decisions that can terminate an iteration of the loop prior to the generation of a movement authority. Either the pod has not confirmed the adjustment sleepers or the reservation has not been extended since the last check. The steps are explained in detail below.

A. This operation returns the first pod that has not been checked yet. If there is no more unchecked pods, the operation will return null.

B. The instance of the pod is stored to be available for the following operations.

C. The decision node will direct the flow to the left if the stored instance of the pod is null. Otherwise, it will direct the flow downwards into the "get first section" block(1).

D. If the "get first section" block doesn't trigger the "ignore" pin, the stored instance of the pod will be retrieved and sent to the "pod" pin in the "reserve sleepers" block(2). The "start section" pin in the same block will receive the returned section from the "get start section" block simultaneously.

E. When the "last available" pin in the "reserve sleepers" block is triggered, the

decision node will check if the current pods reservation has expanded. If this is the case, the green flow to the right will be triggered. If not, the flow is directed upwards.

11.7.2 Generating and Sending a New Movement Authority(green flow)

When the green flow is triggered, it means that the pods reservation has been extended and the pod needs to be made aware of this via a movement authority. If the pod is entering the control area via the switching intersection, a flag will be set in the movement authority to notify the ”switch manager” block. The procedure is explained in detail below.

F. The new movement authority is generated and returned in this operation. A movement authority contains the pods last reserved sleeper as well as the maximum allowed speed level.
G. If the pod is approaching the control area and is going to enter it via the switching intersection, the flow will be directed upwards. If not, the flow is directed to the right and the movement authority is sent directly to the switch manager block via the ”new MA” pin.
H. This operation sets the ”switching required” flag in the movement authority to true before it is sent to the ”switch manager” block via the ”new MA” pin.

11.7.3 Generating and Sending a Handover(blue flow)

If the current pod is given movement authority to the last sleeper in the control area, the next guideway controller needs to be made aware of this via a handover message. A handover message will only be sent once. The procedure is explained below.

I. If the ”reserve sleepers” block returns null via the ”last reserved section” pin, it means that the pod is given movement authority until the last sleeper in the control area, and the flow is directed to the right. If not, it is directed downwards into the flow final.
J. A handover only needs to be sent once. To make sure of this, a flag in the pod object is set to true when it has been handed over to the next guideway controller. If this flag is set to false, the flow is directed to the right. If the flag is set to true, the flow is directed downwards into the flow final.
K. This operation generates and returns the handover. A handover contains the entrance sleeper, which is the first sleeper in the next control area, as well as the pods id. The handover is sent directly to the ”format message” block via the ”handover” pin.

11.7.4 GetStartSection

The ”start section” block will either return a section via the ”section” pin or trigger the ”ignore” pin, which will end the current loop iteration. In the first case, the returned section will be the section where the ”reserve sleeper” block should start the check for available sleepers. The procedure starts when the ”pod” pin is triggered:

1. The instance of the pod is stored to be available for the following operations.
2. If the pod has no previous reservations, the flow will be directed to the right. If it already have a reservation in the current control area, the flow will be directed to the left.
3. This operation returns the section where the last reserved sleeper is located, which will be sent to the ”reserve sleepers” block via the ”section” pin.
4. If the pod has no previous reservations, it means that it is located outside the control area. Before it reserves any sleepers and get a movement authority into the control area, it must have confirmed the reception of the adjustment sleeper info. If such a registration is registered, the flow is directed to the left. In the opposite case, the flow is directed to the right, which will trigger the ”ignor”e pin and end the iteration.
5. If the pod is approaching the switching intersection, the flow is directed to the right. If not, it must be approaching a merging intersections, and the flow is directed to the left.
6. This operation retrieves the section the pod will merge into and sends it to the ”reserve sleepers block” via the ”section” pin.
7. If the pods destination require the pod to switched straight, the flow is directed to the right. In the opposite case, the flow is directed to the left.
8. This operation returns the first section located on the side track.
9. This operation returns the first section located on the straight track.
10. The returned section from either step 8 or 9 is stored in the pods next section variable.
11. This operation returns the switch section and sends it to the reserve sleepers block via the section pin.

11.7.5 Reserve Sleepers

This block is the most essential block in the guideway controller. When the block is activated via the ”pod” and ”start section” pin, the last available sleeper in front of the pod will be returned. If this sleeper is the last in the control area, null will be returned via the ”last reserved section” pin, which will potentially trigger the generation of a handover message in the ”movement authority manager” block. To be able to do this, the block initially need to receive the control area and station name via the corresponding input pins. The yellow flow marks a loop, which is

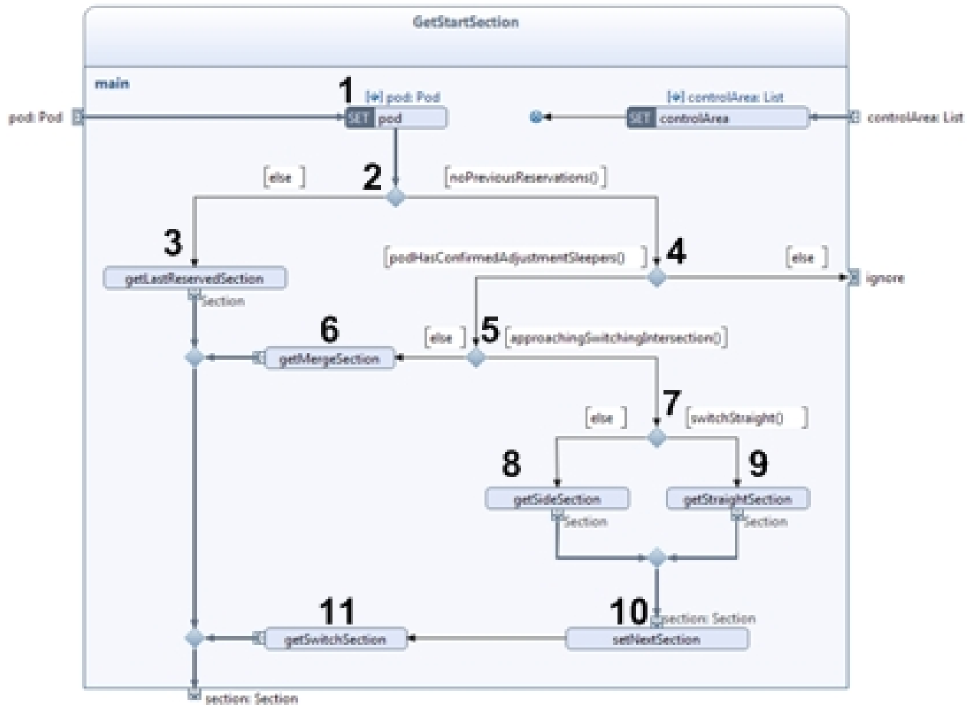


Figure 11.15: The inside of the get first section block. This block either returns the section where the reserve sleepers block should start the reservation procedure or terminates the current iteration of the loop in the movement authority manager block. The procedure is described by an activity diagram.

iterated every time a new section is checked for available sleepers. The procedure is illustrated in section 11.7.5. Figure 11.17 supplements with an example.

1. The instance of the pod and section is stored to be available for the following operations. In figure 11.17, the blue section will be stored with pod A while the green section will be stored with pod D.
2. The decision node will direct the flow downwards if the pod is the last registered pod in this section. If there is a pod registered further ahead in the same section, the flow will be directed to the left, which exits the loop. This is where pod A will leave the third iteration of the loop.
3. If the pod is the last pod in the current section, the last sleeper in the section is returned.
4. If there is a pod further ahead in the section, the sleeper in front of this pod is returned.

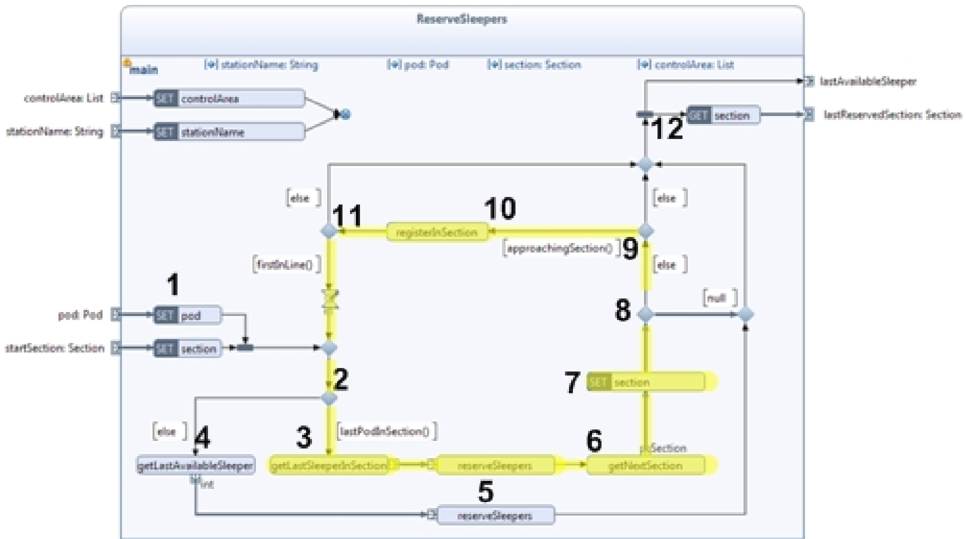


Figure 11.16: The inside of the reserve sleepers block. The block mainly consists of an internal loop(yellow flow) that will iterate one time for each section that is checked for available sleepers.

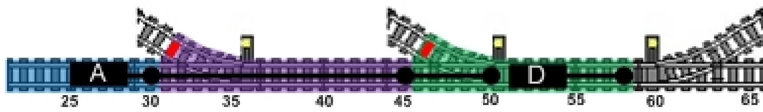


Figure 11.17: A step by step illustration of the reservation process. The vertical lines represents the pods reservations after the process is finished. The dots along the line represents each time the loop passes step 2 in figure 11.16. The different colors represent different sections in the control area. The guideway withouth color is outside the control area. The guideway controller will finish the procedure for pod A before starting with pod D.

5. The reservation is extended to the sleeper number returned by operation 3 or 4.
6. This operation returns the next section along the track. In a switching intersection, this can either be the first section on the side track or the first section on the straight track. If the current section is the last one in the control area, null is returned.
7. In this step, the next section replaces the current section, which was stored in the first step.
8. If the next section is set to null, the flow will be directed to the right. If the next section is located in the control area, the flow will be directed upwards.

This is where pod D will leave the loop in its first iteration.

9. If the pod is approaching the next section, the flow will be directed to the left. The pod must be closer than 20 sleepers for this condition to be met. Otherwise, the flow will be directed upwards. If the current track is the station track and the pods destination is set to this station, the flow will be directed upwards regardless of how close the pod is to the next station. This condition makes makes sure that the pod wont just pass the station platform.
10. This operation registers the pod in the next section.
11. If the pod has first priority, i.e. is the first pod in line at the next section, the flow will be directed downwards and the next iteration of the loop is started. If there is a pod with higher priority, the flow is directed upwards.
12. The last checked section is retrieved and returned via the last reserved section pin. At the same time, the last available sleeper pin is triggered, which means that the block has reserved the last available sleeper for the current pod.

Regardless of the pods positions, the next check will start at the green section for both pods since this is the section of their last reserved sleeper.

Chapter 12

The Pods Internal Functionality

12.1 The Pod Block

The top layer of the pod consists of 6 blocks and some operations. The functionality inside the color sensor is thoroughly explained in some of the previous chapter and will not be a focus in this chapter. Block 1 and 3 as well as all the operations marked by green in figure 12.4 are only active in the in the start-up procedure before the pod is placed at the start track.

Figure 12.1: The top layer of the pod consists of 6 blocks. 1 - calibration, 2 - propulsion controller, 3 and 4 - robust MQTT, 5 - color sensor, 6 - pod controller

12.1.1 The Start-up Procedure(green flow)

A. When the pods program is started, the "calibration" block is activated. This block gives the opportunity to adjust the speed control servo manually by

pressing the buttons on the intelligent brick in order to set the speed level on the battery box to 0. Every time the operator pushes a button, the propulsion controller will be ordered to rotate the speed control servo a certain number of degrees. When the wheel on the battery box is in the exact center position, the operator will push the enter button, which terminates the ”calibration” block.

B. When the calibration is done, the ”propulsion controller” will be notified simultaneously as this operation generates the initial registration parameters. These parameters contains the address of the MQTT broker as well as the topic which the registration response from the central will be published to.

C. When the ”robust MQTT” block has connected to the broker and subscribed to the topic given in the parameters, this operation will generate a MQTT registration message to the central.

D. When the central receives the registration message, it sends a response with the pods id. This operation extracts the id from the MQTT message and returns it. At the same time, block 3 is terminated.

E The pods id is stored for future use in this step.

F The initial parameters are generated in this operation. It will contain the same broker address as in step B, but the topics will be LEMIP/TO_POD/id/# and LEMIP/TO_POD/BROADCAST/#.

G When the ”robust MQTT” block has connected to the broker and subscribed to the topics given in the parameters, the pods id is retrieved and sent to the ”pod controller” block and ”color sensor” block, which will initialize them.

12.1.2 Propulsion Controller Block(2)

This block is in control of the speed control servo, which is connected to the battery box. Initially, the block needs to be calibrated in case the speed level on the battery box isn’t set to 0 at start-up. When the calibration is finished, the ”finished calibration” pin is activated, which resets the tacho counter inside the servo to 0. Because each speed level corresponds to a rotation of 15 degrees, the position of the speed control servo can be set by the formula $15xspeedLevel$. Every time the speed level is changed, the block will return the speed level to the ”color sensor” block after the rotation has finished.

12.2 Pod Controller Block

12.2.1 The Generate Message Block

This block formats and distributes the received messages to the other blocks based on the message topic. It is organized the exact same way as the corresponding block in the guideway controller. However, it does not contain any filter or buffer because the subscribed topics are fixed and it doesn’t matter if one message interrupts the

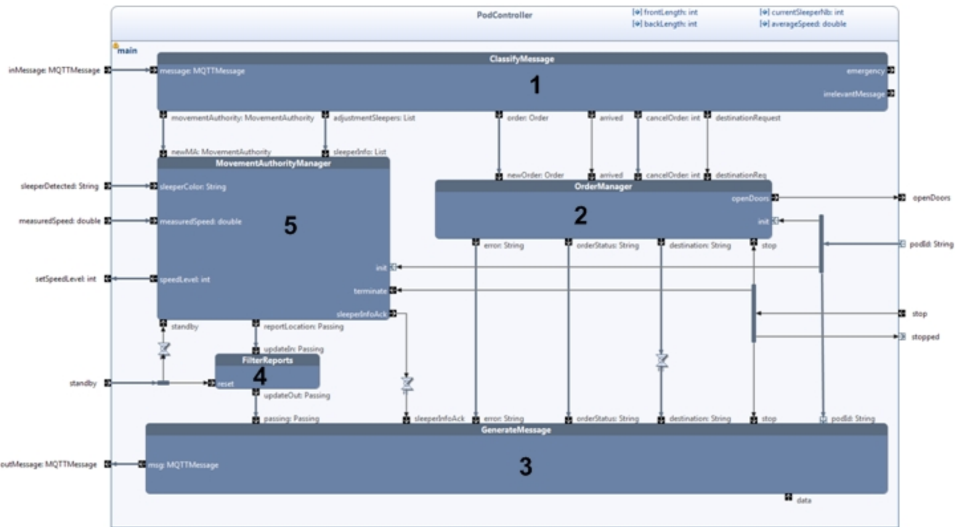


Figure 12.2: The pod controller block consists of 5 blocks. 1 - classify message, 2 - order manager, 3 - generate message, 4 - filter reports, 5 - movement authority manager

processing of a previous one. The block has only one input pin, which is the incoming MQTT messages and 6 active output pins, which either distributes the messages to the ”movement authority manager” or the ”order manager” block.

12.2.2 The Order Manager Block

The order managers main task is to keep track of the pods destination by managing the orders. There are two ways the destination can change. If a new order is received from the central and the pod has no currently active orders, the destination will be set to the departure station in the new order. If a new order is received when an active order is processed, the new order will only be stored in a queue. The destination will also change every time the pod arrives a station. The only exception is if there are no new orders in the queue. In this case, the destination is not changed. The order manager is made aware of a station arrival by the guideway controller via the ”arrived” pin. When an order status is changed, the order manager sends an update to the central.

Orders can also be cancelled, however, an active order cannot be cancelled if there are no orders in the queue. If this is the case, an error will be returned to the central to make sure that all pods on the transit track has an active order and a destination station.

12.2.3 The Generate Message Block

This block receives messages from the other internal blocks and formats them into a MQTT message. The topic is assigned based on the input pin. It is organized the same way as the ”format message” block in the guideway controller.

12.2.4 The Filter Reports Block

Because the ”movement authority manager” block generates a location update every time a sleeper is passed, the ”filter reports” block will only pass every 5th update to the ”generate message” block. This is done to reduce the network traffic and save processing power in the central and guideway controllers. The filter will pass the location updates where the speed is set to 0, i.e. when the pod is reporting standby.

12.3 The Movement Authority Manager Block

The movement authority manager keeps track of the pods position and controls the speed. It has 7 input pins:

init - this pin initiates the block.

sleeperInfo - this message is sent from the guideway controller and contains the information about all the adjustment sleepers that the pod will pass in the control area.

newMA - a new movement authority is sent from the guideway controller every time the pods reservation is extended and contains the new stop sleeper and speed limit.

sleeperColor - every time the color sensor detects a new sleeper, the color of the sleeper is reported via this pin.

measuredSpeed - every time the color sensor detects a new sleeper, the measured speed is reported via this pin. The ”sleeper color” pin is always triggered simultaneously.

standby - when the pod has come to a complete stop, the color sensor reports a standby via this pin.

terminate - this pin is triggered when the program is stopped.

And 3 output pins:

sleeperInfoAck - this is activated immediately after the reception of the adjustment sleeper info from the guideway controller as a confirmation.

reportLocation - every time the color sensor detects a new sleeper, a report containing the current location and speed is sent via this pin.

speedLevel - when the pod needs to change its speed, the new speed level is sent directly to the ”propulsion controller” block via this pin.

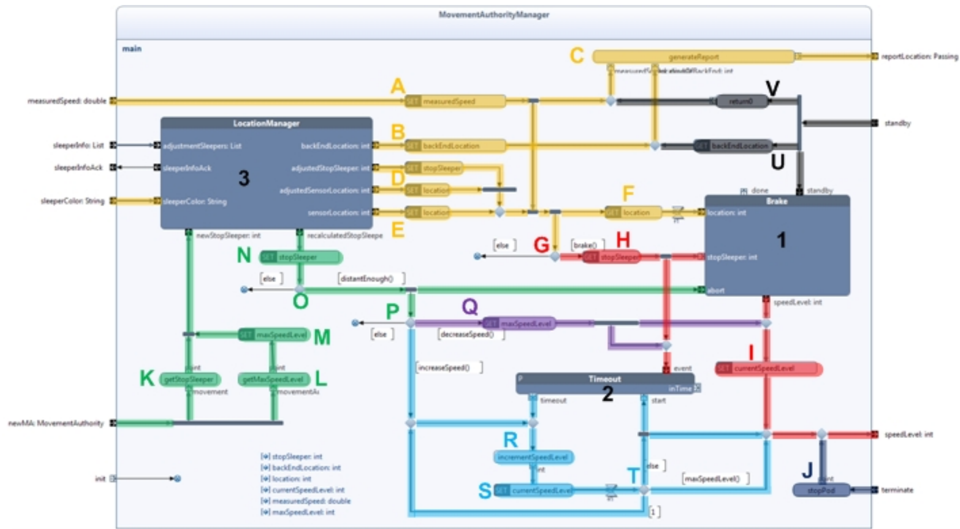


Figure 12.3: Screen dump of the movement authority manager block

12.3.1 Passing a New Sleeper(yellow flow)

Every time the color sensor detects a new sleeper, the yellow flow is activated via the ”sleeper color” and ”measured speed” pin. The sleeper color is reported directly to the ”location manager” block, which will update the position. Every time the yellow flow is triggered, a location update will be generated. In addition, the required braking distance will be calculated based on the measured speed and compared to the remaining distance to the stop sleeper. If the pod is closing in on the stop sleeper, the braking procedure will be initiated. The procedure is explained in detail below.

A. The measure speed is stored to be available for the next operations
B. The location of the back end is stored to be available for the next operations.
C. This operation returns the location update report when it receives both the measured speed and current location.
D. If the detected sleeper was an adjustment sleeper, the stop sleeper is updated and stored as well as the new location.
E. If the detected sleeper was a normal sleeper or a correction sleeper, the updated location is stored to be available for the next operations.
F. the location of the color sensor(stored in operation D or E) is retrieved.

12.3.2 Stopping the Pod(red flow)

If the pod is closing in on the stop sleeper, the speed needs to be reduced gradually. This is handled by the ”brake” block, which is activated by the red flow.

G. Every time a new sleeper is detected, the stopping distance is calculated by the formula described in section 7.1.3. If the pod needs to initiate the braking process, the flow is directed to the right. If not, the flow is directed to the left into the flow final node.
H. The stop sleeper relative the the pods current number series is retrieved.
I. The changed speed level is stored to be available for the other operations.

12.3.3 Stopping the Program(dark blue flow)

When the pods program is stopped, the speed level of the pod is set to 0 in operation J.

12.3.4 New Movement Authority(green flow)

When the pod receives a new movement authority from the guideway controller, the stop sleeper is updated and the pod will adjust the speed to the given speed limit.

K. This operation retrieves the stop sleeper from the movement authority
L. This operation retrieves the maximum speed level from the movement authority
M. The maximum speed level is stored.
N. The ”location manager” block calculates and returns the number of the stop sleeper relative to the pods current sleeper number series. This operations stores the new stop sleeper.
O. The decision node will send the flow to the right if the new movement authority is more than 2 sleepers ahead of the current position. If not the flow is sent to the left into the flow final node.
P. If the new speed limit is higher than the current speed level, the flow is directed downwards into the the blue flow. If the new speed limit is lower, the flow is directed to the right into the purple flow. If the speed limit is the same as the current speed level, the flow is directed to the left into the flow final node.

12.3.5 Reducing the Speed(purple flow)

If the speed limit is lower than the current speed level, the maximum speed level is retrieved in operation Q and sent to the ”speed level” pin.

12.3.6 Increasing the Speed(light blue flow)

When the pod needs to accelerate, the speed level will increase gradually until the maximum speed level given in the movement authority is reached. The ”timeout” block will trigger the speed level increase every 1.5 seconds.

R. This operation increments the speed level by 1 and returns it.

S. The speed level is stored

T. If the current speed level is 1, the flow will be directed downwards and immediately increment the speed level to 2. This is done because the pod won't move at speed level 1. If the maximum speed level is not reached, the flow will be directed upwards, which will send the new speed level to the "speedLevel" pin and start the timer. If the current speed level is the same as the maximum speed level, the flow is directed to the right, and the timer will not be started.

12.3.7 Standby(black flow)

When the pod has come to a complete stop, the "brake" block is initially notified. To report to the central and guideway controller that the pod is at standby, a new location update is generated. Operation U retrieves the current location of the back end of the pod while operation V returns 0, which will be reported as the current speed.

12.3.8 The Brake Block(1)

This block is initiated when the pod is closing in on the stop sleeper and needs to start braking. It makes sure that the pod is stopping smoothly at the intended location by gradually reducing the speed.

The block has 4 input pins:

stopSleeper - this pin initiates the block and includes the current stop sleeper, which is the last sleeper the front end of the pod can pass.

location - every time a sleeper is passed, the this pin is activated and contains the current location of the color sensor.

standby - this pin is activated by the corresponding pin in the higher level "movement authority manager" block.

abort - When a new movement authority is received it means that the reservation has been expanded and the braking process is aborted.

And 1 output pin:

speedLevel - when the speed needs to be reduced, this pin is activated and contains the desired speed level.

When the pod is approaching the stop sleeper, the block is initiated. It will immediately set the speed level to 2. Every time the pod passes a new sleeper, the block will check if it's less than 6 sleepers to the stop sleeper. If this is the case, the speed level will be set to 1. Now the block will check if the it's less than 2 sleepers to the stop sleeper. If this is the case, the speed level will be set to 0. When the pod

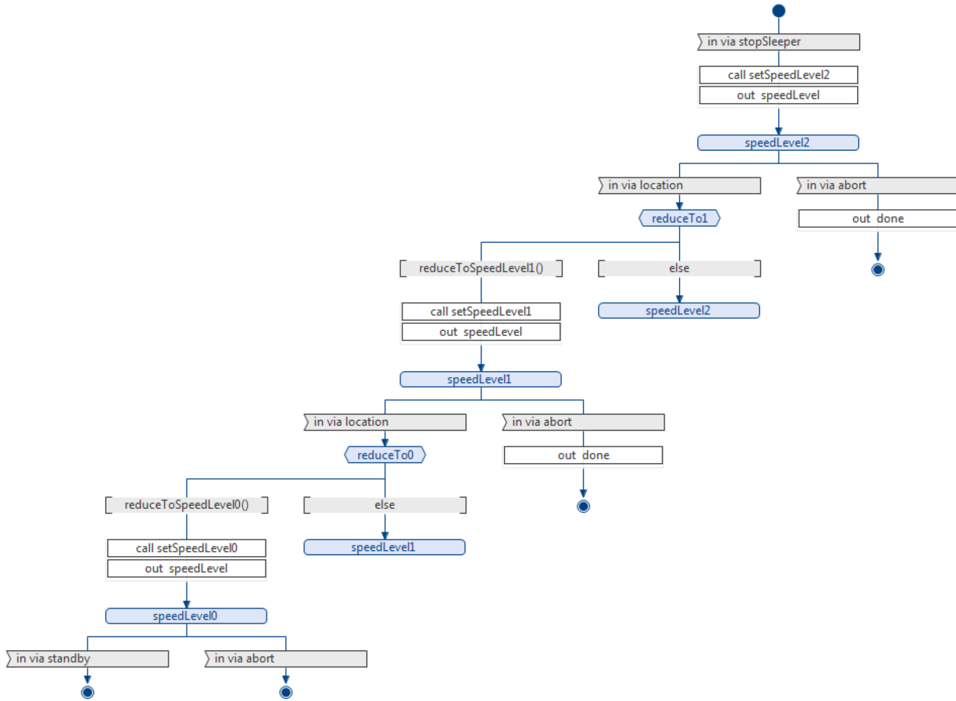


Figure 12.4: The state machine diagram of the brake block. The block has the states speed level 2, speed level 1 and speed level 0, which represents the speed level setting on the battery box.

comes to a complete stop, standby will be reported and the block terminates. If a new movement authority arrives, the braking process will be aborted regardless of the current state.

12.3.9 The Timeout Block(2)

This block is activated when the pod needs to accelerate. It is basically a countdown clock that will activate the "timeout" pin a given time after it is initiated by the "start" pin unless it is cancelled by the "event" pin. The time is set to 1.5 seconds. The block is included in one of the Reactive blocks standard libraries.

12.4 The Location Manager Block

This block is responsible of keeping track of the pods position and calculate the stop sleeper number relative to the current sleeper number series. To calculate the relative stop sleeper, the adjustment sleeper information is needed. The relative

stop sleeper makes it less complicated to calculate the needed braking distance. The block has 3 input pins, where the ”sleeper color” and ”sleeper info” pin contains the same information as the corresponding pins in the higher level ”movement authority manager” block. The ”new stop sleeper” pin contains the retrieved stop sleeper from the last movement authority. The block has 6 output pins:

sleeperInfoAck - this pin confirms the reception of the adjustment sleeper info from the guideway controller

backEndLocation - every time a sleeper is passed, this pin will return the position of the back end of the pod.

sensorLocation - every time a sleeper is passed, this pin will return the position of the color sensor.

adjustedSensorLocation - When an adjustment sleeper is passed, this pin will return the position of the color sensor

adjustedStopSleeper - When an adjustment sleeper is passed, this pin will return the stop sleeper relative to the newly entered number series.

recalculatedStopSleeper - When a new movmement authority is received, this pin will return the calculated adjustment sleeper relative to the current number series.

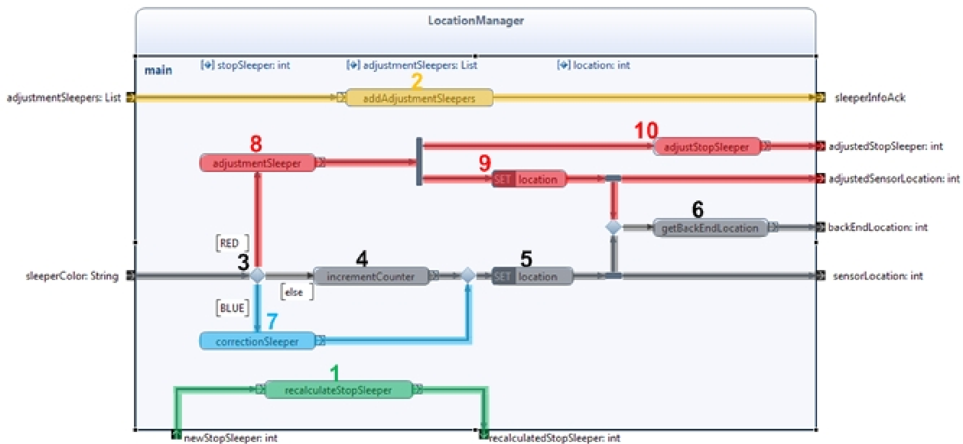


Figure 12.5: The location manager block.

The ”location manger” block is illustrated in figure 12.5 with 5 different colors on the flows.

The green flow is activated when a new movement authority arrives from the guideway controller. The flow is triggered by the ”new stop sleeper” pin. Operation 1 recalculates the stop sleeper based on the current number series the pod is currently in. Let imagine that the the current number series is 300-400 and the next one is

500-600. If the stop sleeper in a new movement authority is 550, the recalculated stop sleeper would be 450. Similarly, if there is one more number series in the control area spanning from 100-200 and the stop sleeper in the movement authority is 150, the recalculated stop sleeper would be 550. The recalculated stop sleeper is returned via the ”recalculated stop sleeper”’ pin.

The yellow flow is activated when the guideway controller sends the adjustment sleeper info. The flow is triggered by the ”adjustment sleepers” pin. Operation 2 adds the new adjustment sleeper info to the list of adjustment sleeper info before a confirmation is sent via the ”sleeper info ack” pin.

the grey flow is triggered via the ”sleeper color” pin every time the color sensor detects a new sleeper. Depending on the color of the sleeper, the decision node(3) will direct the flow upwards if it’s a red sleeper, downwards if it’s a blue sleeper and to the right if it’s a grey/normal sleeper.

If it’s a grey/normal sleeper, operation 4 will return sleeper number/location incremented by 1 before the location is stored in operation 5. The new location is returned via the ”sensor location” pin. At the same time, operation 6 will calculate the back end of the pod. This location might be in another number series than the sensors location.

If the new sleeper is red, it’s an adjustment sleeper, which means that the pod has entered a new number series. Operation 8 will return the adjusted sleeper number, which is the first sleeper in the new number series. In operation 9, the location is stored before it’s returned via the ”adjusted sensor location” pin. At the same time, the grey flow and operation 6 is triggered, which calculates and returns the location of the pods back end. In addition, operation 10 will adjust the stop sleeper to be relative to the new number series. The adjusted stop sleeper is returned via the ”adjusted stop sleeper” pin.

If the new sleeper is blue, it’s a correction sleeper. Operation 7 checks the location via a modulo operation and corrects the sleeper number if it’s offset. The corrected location is returned before the blue flow is merged into the grey one and the new location is stored in operation 5.

Chapter 13
Testing and Evaluating the System

This first part of this chapter describes the limitiations in the lab due to lack of time, space and lego bricks. The second part describes the lab setup while the third part evaluates the system performance.

13.1 Limitations

Because of confined space and lego parts, I wasn't able to build the guideway described in chapter 4. In addition, I didn't have enough time to make a fully automatic central and dynamic routing. However, I was able to build a guideway with enough intersections and stations to test most possible situations. By making a graphical user interface allowing an operator to generate orders to the pods manually, it was possible to make the pods move continuously between stations.

13.1.1 Reduced Guideway Size

The guideway layout is limited to 4 stations and 2 transit track intersection pairs. The main reason for this is lack of space and Lego parts because another group was using the same lab and equipment for another thesis. However, the current guideway layout contains enough possible paths to create interesting situations. The layout is illustrated in figure 13.1. The control areas and number series are divided using the same principles as explained earlier. This implies 6 control areas, one for each switching intersection.

Finally, I have added a start track. This is necessary to move a newly started pod into the system. Every pod needs to be put down on the start track after it has been started and calibrated. When it registers to the central, it will be assigned an id and handed over to the guideway controller responsible of control area 1, which will assign it a movement authority if there are no pods approaching the merging intersection from the transit track.

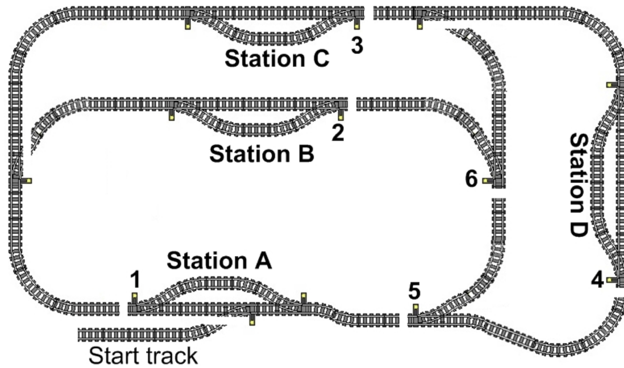


Figure 13.1: Because of confined space in the lab and lack of lego bricks, I had to reduce the guieway desgin to four stations and a total of six intersections. The control areas are illustrated with a gap and numbered at the start of each control area.

13.1.2 Manual Central

Unfortunately, I did not have enough time to make a fully automatic central supporting dynamic routing. Instead, static routes are used. As a consequence, the routing tables will only be based on the length to each station along the different paths instead of the actual travel time. This means that there are no mechanism to route pods around congested track sections or sections blocked by a stalled pod.

In addition, the central does not support automatic order generation. Instead, I have implemented a graphical user interface (GUI), where orders to a specific pod can be generated. The GUI includes a checkbox that can be used to set the order to be repeated until a new order is generated. This causes the central to send the same order to a pod as a response when it reports the previous order to be finished.

I think manual order generation is initially a better solution because it makes me able to control the paths of the pods. This makes it possible to create situations where one pod has to yield for another in merging intersections. By issuing the same order to pods at the same station at the same time, it is also possible to check the actual length of the headway by observing how close the pods will move.

The most obvious missing function for observers is the reallocation of idle pods at station tracks. When a pod has finished all of it's assigned orders and is not immediately assigned a new one, it will result in all the consecutive pods to be blocked. Initially, only pods arriving the same station will be blocked, but as the

station track fills up, pods that is supposed to pass the station along the transit track will also be blocked when the queue of pods reach the transit track. However, this situation will never occur as long as the orders to each pod is repeated continuously.

13.2 Test Setup

Because there are 6 switching intersections, two EV3 intelligent bricks are needed as switch operators because each brick only has 4 outputs. Each of the four switch operators are assigned a button on the intelligent brick. When this button is pressed, the tracks will be switched. This is necessary because there are no sensor to check which position the tracks are in at system startup. By default, the switch operators expect the tracks to be switched to the side. If they are not in this position at system startup, the tracks needs to be switched manually before the first pod is initialized. The switch operators are powered by a battery eliminator, which allows them to always be on. This drastically reduces the number of times manual switching is required because the switch operators only needs to be stopped when the lab isn't expected to be used for a long while.

When a pod is initialized it is given an order to go to station C. It will stay at this station until it is given a new order. In retrospect, I realize that a better solution would be to initially order it to go to a random station. However, this can easily be implemented.

13.3 Evaluating the Performance

After some initial tests, it was obvious that even if the system worked as expected for some time with 3 pods, the chance of unexpected situations increased for every pod in the system. There are mainly two problems occurring repeatedly requiring an interaction, such as system restart or removal and deletion of a pod.

13.3.1 The Pod Reports an Incorrect Location

Sometimes the pods are moving too slow because the batteries powering the propulsion motor is run down. As a result, some sleepers are detected twice by the color sensor because the expected speed doesn't correspond to the actual speed. As a result the pod will report an incorrect location, potentially causing crashes. However, at adjustment or correction sleepers, the location will be corrected. The real problems occur if a pod detects an adjustment sleeper twice, because the pod will think it is located at the next adjustment sleeper. This is a more severe mismatch between the reported location and the actual location.

Every time the pod detects an adjustment sleeper, it expects to have received the information about it. If it cannot find any information about the adjustment sleeper, the pods control program will encounter a critical error and shut down. If the pod is moving when this happens, it will continue along the tracks until it crashes into another pod or is removed. The same situation can occur if a pod has missed enough sleepers to think it's behind it's actual location and violates its movement authority. If this results in the pods color sensor to cross an adjustment sleeper it shouldn't cross, the program will shut down.

By making sure the batteries are always charged, the source of the problem is removed. To make the system more robust, it's easy to implement a check that will ignore doble detection of adjustment sleepers. To avoid that a the pod continues uncontrollably when it detects an unexpected adjustment sleeper, it can be programmed to stop instead. However, since it's reported location is wrong, it still poses a threat to other pods and needs to be removed from the guideway.

13.3.2 The Pod Misses Important Messages

By using the TCP protocol on the network layer and QoS 2 on the MQTT messages, I assumed that it could be guaranteed that messages is received by all subscribers. However, sooner or later one or more pods in the system will stop because the pod controller block does not receive an essential message, which causes a deadlock. This is usually the sleeper information message sent from the guideway controller when the pod is approaching a new control area. As a result, no confirmation is returned to the guideway controller, which is a requirement to issue a movement authority to the pod. Consequently the pod will not move while it's waiting for a new movement authority.

I suspect that a message arriving immediately after the reception of the sleeper information message interrupts the processing of the sleeper information. I experienced the same problems when not utilizing the buffer in the guideway controller, but I assumed that because the pod receives relatively few messages compared to the guideway controllers, it wouldn't be a problem. For this reason, I didn't implement a message buffer in the pod. I'm quite sure that the messages are received by the MQTT block, but unfortunately, I didn't have time to check this in the lab. However, I had a client subscribing to all the topics in the system, which received the sleeper information. This confirms that the guideway controller publishes the message.

The missed messages might have another cause I haven't thought about, but I'm not able to provoke the same situations in a computer simulation. The simulation I use is simply running the guideway controller program normally and replicate pods running along the guideway by generating location updates from another computer.

This proves the essence of having access to a lab with physical components generating situations that is hard to provoke in a computer simulation.

If I had more time in the lab, I would implement a functionality where the guideway controller would resend the adjustment sleeper information until it receives a confirmation. This would require the pod to disregard duplicated adjustment sleeper information messages, but this is a simple fix. Another solution, which could remove the cause of the problem, is to add a message buffer similar to the one in the guideway controllers. This would make sure that all the incoming messages is processed. Implementing both solution could result in a guarantee that the pod receives and processes the messages, which would decrease the chance of deadlocks.

13.3.3 Expected Behaviour at Normal Operation

When no situations occur where the pods control program shuts down or messages are missed, the system is behaving as intended. The pods are moving smoothly along the guideway and is switched to their destination station according to the static routes. At merging intersections, the pod with lower priority yields for the pod with highest priority. When the first pod has passed, the pod with lowest prioirty follows the first pod at a safe distance. The handover between control areas and switching procedure is usually finished before the pod is closing in on the switching intersection, and it enters the intersection smoothly without any unnecessary stops. When a pod is closing in on another pod, it slows down and stops 1-4 sleepers prior to the pod in front.

Sometimes there is a delay in handover, movement authority or switch command messages. This causes the pods to stop and wait for a new movement authority at the given stop sleeper, which is usually in front of an intersection. This happens relatively rarely, but when it happens, it's usually on stretches with short distance between the different control areas, requiring many consecutive handovers. This is a result of the confined space in the lab. If I had more space, the stretches would be much longer, and the handover procedure would be finished long before the pods needs to slow down.

The headway is a crucial aspect in a PRT system and can potentially make the capacity utilization too low to be affordable. In the model, the headway is approximately 1.5 seconds, which would be really good in a normal sized PRT system. However, in 1.5 seconds, a pod has time to move 60 cm, which is a substantial part of the total length of the guideway. Because a pod at standby needs some time to accelerate to the same speed as the pod in front, the distance is additionally increased. As a result, a guideway in the scale I have used in the lab will start to be congested with more than 4 pods. To simulate more pods without congestion, more space is needed. Nevertheless, the headway is approximately twice the length of a

pod, which isn't a bad relation at all. The problem is rather the relation between the length of the pods and total length of the guideway.

13.4 Using a Loosely Coupled System Design

Coupling refers to how dependent the different units are of each other to accomplish their separate task according to the requirements. The LeMiP system is categorized as a low or loosely coupled system, which means that the different units can solve their assigned tasks independent of the other units. This might sound unreasonable because I have just described scenarios where the system will stall even with one faulty unit. However, the different units will still be able to accomplish their tasks even if they lose the ability to communicate with other units. Further, the whole system will not need a restart when the source of error is removed, but simply carry on as normal.

Let's say that the central is fully automatic and goes off line. This will prevent new orders from being distributed to the pods and routing tables to be updated. Nevertheless, the Guideway Controllers will still be able to guide the pods safely to their destination with the old routes. The chosen route might not be the optimal one, but there are no increased danger of accidents. This means that passengers already embarked in a pod will not notice if the central goes down. When the central is back online, the system will function as normal without any manual adoptions. If the LeMiP system was a real world system, it would be easy to add an extra standby central that would detect if the main central was off-line and go operational. If all the data is constantly shared between the two centrals, no other units in the system would even notice the malfunctioned central.

If one or several of the Guideway Controllers malfunctions, the pods will still not be in any danger of crashing because they are only given authority to move to a certain point, which it will do without any further communication with the Guideway Controller. It will not be able to move any further, but remember that the pods responsibility is only to move to the point it is told to. However, since all the guideway controllers run the same program, it can easily be replaced by a service technician, receive the positions of the pods in its control area and start issuing movement authorities as normal. When I stopped one of the guideway controllers, the pods stopped right in front of it's control area. However, I did not have time to implement a functionality that informs a newly started guideway controller of pods located in it's control area.

If a pod looses connection or fails in any other way, the system will react by defining the area between it's last known position and the point it was authorized to move to as unsafe for other pods. If the central was automatic, this would result

in updated routing tables excluding this section, and the system would continue to work, but without access to the stations that rely on the track section blocked by the faulty pod.

Currently, a faulty pod can be lifted from the tracks and deleted from the system, which will make the previously unsafe area safe again. This will make the system work normally with one pod less. If LeMiP was a real world system, the solution would be similar if a service technician couldn't fix the problem on-site. Since it is much harder to remove a pod from the tracks in a real world system, the first option would be to drive the pod manually to a service track if possible. If not, the pod would be removed from the tracks by a crane and deleted from the system.

Chapter 14
Conclusion and Recommendations for Future Work

This chapter presents the conclusion and recommendations for future work.

14.1 Conclusion

This section presents a brief summary of the process of designing, building and testing the Lego PRT system. The first subsection summarizes the functionality and behaviour of the pods. The second subsection summarizes the design and performance of the control system.

14.1.1 Develop Lego Pods With Realistic Behaviour

Lego Mindstorms is a relatively cheap, versatile and advanced tool to make programmable robots. In combination with the Lego City train, it is indeed possible to build vehicles that can simulate pods in a real world PRT system. In the first part of the thesis, I have described the process of determining the capabilities of Lego and use the discovered features to build models Lego of pods.

The pods can be categorized as “stupid” in the sense that they will only do as they’re told by a higher level control system. The pods are controlled by assigning them *movement authorities*, which contains the location they must move to and at which speed. When receiving a *movement authority*, the pod will accelerate gently to the given speed limit. While moving, it will constantly check its speed and position and start to brake when it’s closing in on the location given in the *movement authority*. The pods will stop smoothly at the given location with an accuracy of 10 cm. The pod determines its speed and position by counting sleepers. Each sleeper is detected by an on board Lego Mindstorms EV3 color sensor directed downwards. The color sensor is able to distinguish the color of the sleepers and the color of the surface between the sleepers.

Making the data from the color sensor sufficiently accurate has been a time-consuming effort. I haven't found any other examples where the color sensor has been used to measure speed and location in a train system. Consequently, I have been obliged to interpret raw data from the color sensor and create a program able to process the data into accurate speed and position measurements. When I tested the location accuracy, the pod was passing approximately 2300 sleepers where only 1 sleeper was missed. This corresponds to deviation of 3,2 cm after travelling 80 meters. The accuracy of the speed measurements depends on the speed. The lower speed, the more accurate the speed measurements are. With a maximum speed of 0.6 m/s, there are no consecutive measurements with a deviation of more than 0,03 m/s. These tests were run on a guideway without intersections. The intersections design causes error readings, and the number of error readings increases drastically when a pod crosses intersections.

While real world PRT systems use multiple sensors to keep track of the pods position, the Lego pods must solely rely on the information given by the color sensor. As a result, the pods location is exposed to the error readings from the color sensor, requiring a way to correct the errors before the mismatch between the actual and perceived location increases to a critical level. In the Lego system, this is solved by attaching colored bricks to some of the sleepers. While this way of correcting the location works most of the times, it also causes the system to encounter critical errors in some situations where colored sleepers are detected twice. These situations occur more frequently when the batteries feeding the propulsion motor in the pod are run down and the pods are moving too slow. I believe that these situations can be avoided with relatively little effort by adding minor changes to the pods program and make sure the batteries are fully charged.

The Lego pods communicate with the higher level control system via the MQTT protocol over a WiFi network. Sometimes important information is lost before it is processed by the pods. As a result, a deadlock occurs because the control system is waiting for an answer from the pod while the pod is waiting for a message from the control system. Because MQTT theoretically can guarantee messages to be delivered to the recipients, I suspect that the messages are somehow lost after it has been received by the pod. Despite of its name, Message Queue Telemetry Transport has no built-in functions for queuing messages. This might cause messages arriving at short intervals to interrupt the processing of the previous one, which I consider the most likely cause of lost messages. This can be resolved simply by adding a message buffer in the pod or implement a functionality in the higher level control system that will detect lost messages and resend them.

All in all, the pods are behaving as intended most of the time. One pod with fully charged batteries can drive around the guideway and stop accurately at the

designated station platforms for more than 10 minutes without any problems. The problems usually appears in the form of deadlocks when the number of pods increases. By making sure that all messages are received and processed by the pods, unexpected behaviour can be reduced to a minimum.

14.1.2 Develop a System to Control the Pods In a Safe and Efficient Manner

I have chosen to design the higher level control system to consist of *embedded* and *distributed* components to make it scalable and robust. Embedded system design refers to systems where the different components are assigned dedicated functions and the different components must cooperate in order to make the system fully operational. To achieve this, I have created three different units that are assigned different responsibilities.

The *switch operator* is responsible of making sure that the tracks in a switching intersection are positioned in the correct direction. It can switch between the two directions by controlling a motor, which is mechanically connected to the intersections switch handle. The switch operator will only switch the tracks when it receives an order to do so. If the tracks are already in the intended position, no further action will be taken.

The *guideway controller* is responsible of all the movements on the guideway. This is the only unit that can issue *switch orders* to the *switch operator* and *movement authorities* to the pods. As a result it's in full control of all the actions on the guideway and is responsible of all the safety related decisions. To order the *switch operators* to switch the pods in the correct direction, the *guideway controller* has access to routing tables where the direction to all the different stations are listed. To be aware of the pods destination station, the *guideway controller* sends a request to the pod when it's approaching a switching intersection.

The *central* was intended to be fully automatic, which would include automatic order generation and distribution to the pods, dynamic routes based on actual travel time and reassigning idle pods to avoid obstruction of assigned pods. Unfortunately, I did not have enough time to implement a fully automatic *central*. Instead, I have implemented a graphical user interface making it possible to manually distribute orders to the pods. When assigning an order, it can be set to repeat indefinitely, which will make sure the pods are always assigned orders and moves along the tracks.

Dynamic system design makes the system scalable by dividing the guideway into different *control areas* and make one *guideway controller* responsible of exactly one *control area*. Similarly, one *switch operator* is responsible of exactly one switching intersection. When the guideway is expanded, the distributed design makes sure

that the load on each control unit stays the same while the number of control units are increased instead. In addition, this have a fortunate effect on the system error tolerance because one faulty unit will only affect the guideway sections it is in control of while the rest of the sections will be operational.

The *guideway controller* is the most crucial unit in the control system. While I designed it, I ran a number of computer simulations to test how it performed. When I had resolved all the errors situations I managed to produce in the computer simulation, I tested the control program in the lab. Even though the program appeared to be flawless in the simulations, a lot of scenarios I hadn't included in the simulations occurred, exposing vulnerabilities in the control system that would otherwise not be uncovered.

After fixing the most obvious errors, the system worked with up to three pods at the same time. The pods avoided crashing and was routed the shortest way to their destination stations. Now and then one or more of the pods failed. However, based on the results while the system was operating normally and the assumption that the most severe errors are easy to fix, I believe that it is indeed possible to simulate a real world PRT control system by using Lego Mindstorms. Although I am convinced that a Lego Mindstorm simulation of a real world PRT system in itself is not a mean to make the project successful, it might be useful as a supplement to computer simulations. In addition it might give investors and politicians a more definite and unambiguous impression of the system than a computer simulation.

14.2 Recommendations for Future Work

Besides fixing the errors I have pointed out previously in this chapter, there is some interseting features that can be added to the system in order to improve it.

Because the pods only rely on the color sensor to determine their position, the most obvious improvement would be to add more sensors to supply location data. Since the pod uses WiFi communication and has a bluetooth tranciever, it would be intereseting to determine if it's possilbe to place beacons at fixed locations along the track and use them to calculate the position of each pod.

A hot topic in the world of ITS is platooning, which means that one vehicle in the lead will heard a line of following vehicles. This could be implemented in the pods control program and tested in the lab.

References

[1] K. M. Overskeid, "Its using lego mindstorms," Norwegian University of Science and Technology, year = 2014, page =12, Tech. Rep.

[2] J. E. Anderson. Some lessons from the history of personal rapid transit (prt). [Online]. Available: http://faculty.washington.edu/jbs/itrans/history.htm

[3] Wikipedia. 1950s american automobile culture. [Online]. Available: http://en.wikipedia.org/wiki/1950s_American_automobile_culture

[4] A. T. Association. Prt2000. [Online]. Available: https://www.advancedtransit.net/atrawiki/index.php?title=PRT2000

[5] ——. Cvs prt. [Online]. Available: https://www.advancedtransit.net/atrawiki/index.php?title=CVS_PRT

[6] Wikipedia. Personal rapid transit. [Online]. Available: http://en.wikipedia.org/wiki/Personal_rapid_transit

[7] XmasGiftIdeas. Christmas gifts for kids. [Online]. Available: http://www.xmasgiftideas.org/shop/lego-mindstorms-ev3-31313/

[8] Wikipedia. Osi model. [Online]. Available: http://en.wikipedia.org/wiki/OSI_model

[9] I. S. Institute. Transmission control protocol. [Online]. Available: http://tools.ietf.org/html/rfc793

[10] Wikipedia. Transmission control protocol. [Online]. Available: http://en.wikipedia.org/wiki/Transmission_Control_Protocol

[11] MQTT.ORG. [Online]. Available: http://mqtt.org/

[12] L. Zhang. Building facebook messenger. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920

[13] E. Xiao. Introduction mqtt. [Online]. Available: http://www.slideshare.net/ericssonxiao/introduction-mqtt-en

[14] Wikipedia. lejos. [Online]. Available: http://en.wikipedia.org/wiki/LeJOS

[15] ——. Java version history. [Online]. Available: http://en.wikipedia.org/wiki/Java_version_history

[16] "Cubestormer 3 smashes rubik's cube speed record," https://www.youtube.com/watch?v=X0pFZG7j5cE, online; accessed 18.02.2015.

[17] Bitreactive. Send email tutorial. [Online]. Available: http://reference.bitreactive.com/tutorials/send-email.html