Dumpsty

P5 Project
Group SW510E16
Software
Aalborg University
21st Dec 2016

AALBORG UNIVERSITY
STUDENT REPORT

Fifth semester at
Department of Computer Science
Software
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK
http://www.cs.aau.dk/

Title:

Dumpsty

Project:

P5-project

Project period:

September 2016 - December 2016

Project group:

SW510E16

Participants:

Christian Dannesboe
Frederik Børsting Lund
Karrar Al-Sami
Mark Kloch Haurum
Lasse Lyngø Nielsen
Søren Lyng

Supervisor:

Bent Thomsen

Pagecount: 73
Appendix range: 26
Published 21-12-2016

Synopsis:

This project is about the design and implementation of a trash bin which is to catch the trash thrown at it. This is done using a Arduino Mega 2560 for the trash bin and a Microsoft Kinect for detecting and tracking the thrown trash.

The report covers the analysis of the problem and hardware, a design specification, system design and implementation. The system is then tested and the results are discussed and concluded upon.

Preface

This report is part of the fifth semester project made by project group SW510E16 at Aalborg University, Software Engineering starting from the 2nd of September to the 21st of December 2016.
The project is based on the *Aalborg-model* study method, where problem and project based learning is the focus. The theme of this semester is to make an embedded system with real-time constraints. The project group chose to make a trash bin that should be able to catch the trash thrown at it.

The project group would like to thank the project supervisor Bent Thomsen, for his support and help throughout the project.

Signatures

Christian Dannesboe	Frederik Børsting Lund	Karrar Al-Sami
Mark Kloch Haurum	Lasse Lyngø Nielsen	Søren Lyng

Reading guide

Throughout the report sources are referred to by the Harvard citation method. When a source is listed in the report, the last name of the author and a publication year is listed. The sources are all listed alphabetically in the ***Bibliography*** chapter.
This is an example of a source listed in the report: ***[Regehr, 2006]****.*
The source may refer to either the whole section or to only that sentence. The way this differs depends on the placement of the dot. If the dot is after the source, then that source refers to the sentence and if the dot is before the source, then the source refers to the whole section.

When referring to figures, tables and source code, numbers are used. Depending on which chapter and number of figure/table, the number is defined.
This example can be used: In chapter X we want to refer to the second figure. This is done by giving the figure number ***X.2****, where X is the number of the chapter we're referring to.*

When referring to source code, we're using code snippets. These code snippets aren't necessarily the full source code, but may be shorter version of it and/or missing comments. When code has been removed in the snippets, the use of three dots are used: "...", these dots show that some code altering has been made in the snippet, whether it's because it's long and irrelevant for understanding the purpose of the code, or because we're simply just trying to explain those few lines.

Throughout the report requirements are split into four colours with four different meanings. The colour blue refers to new requirements or focus requirements for that specific increment. The colour green refers to a requirement being fulfilled. Orange means that the requirement is fulfilled but has been changed. Red means that the requirement has been changed, but not yet fulfilled.

Contents

List of Figures

Listings

1 Introduction

An embedded system is a computer system which only has a few functions. It is embedded as a part of a whole device, which then also includes hardware and/or mechanical components. Embedded systems are everywhere in our everyday lives. The range for embedded systems could be all from saving lives with pacemakers to fun gadgets. [Techopedia.com]

An embedded system as a gadget, is a technological and small object or apparatus, which has a certain functionality. This functionality is limited to a certain niche.
In this project, the embedded system in consideration will be designed as a gadget, with the sole purpose of catching trash thrown at it. A trash bin that can catch trash will make the act of cleaning more fun and interactive, and the availability of each individual trash bin will be tremendously increased, because it will be able to move around.

A similar product has already been developed by Minoru Kurata, called "Smart Trash Can", the trash bin has won an Excellence Award at the Japan Media Arts Festival. One of the downsides of this product is that the catch-efficiency is very low, round 10-20%, according to Minoru Kurata. [Aamoth, 2013]

For this gadget to be usable, it must be designed as a real-time system, as this system will have certain deadlines for each task to be executed in time. The trash bin must be able to identify an object coming towards it, make some computation to identify a point to catch it from, and then compute a path to catch the object. All this must be done before the object lands on the ground, which makes the use of real-time system design a part of this project. A real-time system is a software system which is subject to a real-time constraint, and the system must control and affect an environment, by receiving data and process these, within a certain time limit.

1.1 Problem statement

Based on the introduction above, the problem statement for the project is as follows:

How can an embedded system control a trash bin to detect, track and catch a thrown object within a designated area?

Process model 2

Process models for software development is a representation of the process and the activities required for developing the software. In software engineering there is two different development approaches, pland-driven and agile development. The plan-driven approach is based on sperate development stages and iterations can occur within the activities e.g. the implementation of the software. The agile approach the specification, design, implementation and tests are all iterated on as a whole. [Sommerville, 2016]

The process model is a meld of elements from both plan driven and agile development. The process is plan driven in so far as to include a clear goal of what the initial requirements of the system are. The process model is very agile in that, although there is a clear outline as to what the requirements are, these are incrementally approached, with a simple starting point becoming increasingly covering of the project vision through the subsequent increments. The reason the group approached this project with our 'own' process model is to better explain what we're actually doing, rather than forcing ourselves into picking a development method and try to explain what we're doing according to that method.

The increments will be split up into four sections. Initially some of the already known requirements will be considered from previous increments and for the first increment from the analysis chapter, and these requirements will be the topic of the increment. After the initial requirement consideration, a design phase follows which details the design and problem considerations preceding the implementation. The design phase is then followed by an implementation phase which will explain the implementation process. The final phase is an evaluation of the increment and will concern whether the requirements were fulfilled, if an requirement needs to be altered, or if an requirement will be continued in the following increment. The evaluation will also include several tests of the implementation, in order to see what works and what doesn't work. The evaluated requirements from an increment will then become the starting point of the next increment. An illustration of the process model described can be found in figure 2.1.

The process includes both pair programming and pair writing, which is to promote cooperation and attempt to avoid having a single individual involved in a task beyond their capabilities and to get a second opinion. This is also to ensure that information is quickly communicated and exchanged by group members. The process uses pair programming because it promotes better programming and is usually very motivating in socially engaging issues with cooperation of project members.

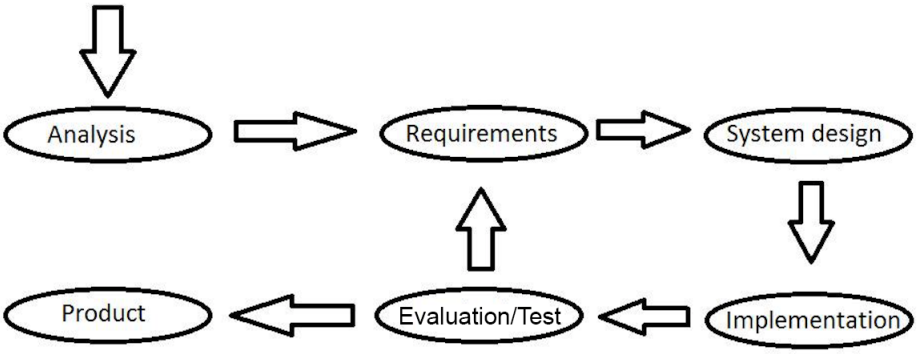

Figure 2.1. Process model used for the project

Toolchain 3

The following tools has been used to create the report and the product of the project. Every tool in the toolchain has a description stating the benefits and use of the tool.

- GitHub
- TeX Studio
- Visual Studio
- Arduino IDE 1.0.3
- BoundT
- UPPAAL
- ArduinoUnit

Use of tools

GitHub is used with the intention of merging and sharing documents. This tool is also used here to verify each version of the documentation and code, to ensure that everyone has the latest update of the project.
TeX Studio is used as a writing environment when writing with the markup language LaTeX. After some experience it has proven to be a great tool for creating a report. This tool in concoction with GitHub makes it possible to work on the project in pairs or alone, with little merging conflicts.
Visual Studio is used in this project to create the C# code for the Kinect.
Arduino IDE 1.0.3 is used in this project to code the Arduino. Any later IDE is not suitable for this project, which will be explained in the report.
BoundT is used to calculate the bounds of the code, which will be used in a WCET-analysis. [Tidorum Ltd]
UPPAAL is an integrated tool environment for modeling, validation and verification of real-time system. [Uppsala University og Aalborg University]
ArduinoUnit is a unit test framework for arduino projects and is used for the unit tests in the tests chapter, 8, of this project. [MacEvoy et al.]

Analysis 4

From the analysis chapter it should be possible to derive the requirements for the smart trash bin, Dumpsty. The initial user story will help make the user requirements for the project. After the user requirements has been formed, the information from the user story will be analysed in greater detail, depicting three different phases of the process: Throwing, detecting/tracking and catching. These three phases will be the inspiration for the system requirements. The analysis chapter will end up with a problem statement for the project.

4.1 User story

Benjamin is a software engineer who is tired of wasting his precious time on the job with walking back and forth from the trash bin in his office, and therefore wants to be able to throw his trash in the general direction of the trash bin instead, without missing the trash bin. He often has to pick up the trash after throwing it at the trash bin.
Benjamin wishes that the trash bin could move and collect the trash for him, so that he can throw his trash in the general direction of the trash bin, and the trash bin could then place itself in a way, that allows it to catch the trash before it lands on the floor. This would optimize the time Benjamin uses each day on collecting the trash he did not land in the trash bin.
If Benjamin throws at the trash bin from a designated side of a sensory camera, the robotic trash bin should identify the trash, move the trash bin to a place where it would be able to catch the trash, before it hits the ground.
If Benjamin throws outside a designated area of the robotic trash bin, it should not try to catch the trash, as it would compute that it is not able to get to the point of impact before the trash hits the ground.
If Benjamin and another person from his office throws trash to the trash bin at the same time, it should prioritize the first identified trash.

4.2 User requirements

The user requirements of the project have been conducted from the section 4.1. User requirements are simple requirements written in a natural language, which should help everyone get an understanding of the requirements for the project.

- The trash bin must be able to move itself to the colliding position of the trash within a certain time limit, with a certain precision.
- The trash bin should only consider trash thrown within a certain area of itself.

4.3 System capabilities

This section will define the required capabilities of the system, which will include the functionality and hardware needed to fulfill the tasks of the embedded system. This is divided into three different categories, which are the subsections of the section. These categories, along with the user requirements, will define the system requirements for the project.

Throwing
When considering the process of throwing the trash, certain requirements must be met, which will be simplified for this project: The trash, from here on referred to as an “object", must be a round object of some sort, in this project a table tennis ball will be used.

Detecting and tracking
Detecting and tracking the object are two very essential tasks, but are considered as one. For detecting the object, a sensor should be able to recognize the specific object.
When the object has been detected, it will then be tracked by the same sensor. The sensor is going to track the direction the object is heading and the speed of the object. This will be used to calculate the point of impact between the object and the ground.

Catching
The catching phase is when the robot calculates the impact point, and moves to that specific point. The robot gets information about the thrown object, in this case the impact point, from the sensor and uses the given information to calculate the impact point. What data the robot gets from the sensor and how the calculation is done, will be explained in later chapters.

4.4 Hardware

The following subsections will describe the hardware considerations for the project, the hardware's capabilities and it's limitations, to specify how the hardware can meet the requirements.

4.4.1 Sensors

Sensors were used to measure the environment around the Arduino. The sensors in consideration are described in individual sections.

Microsoft Kinect

The Microsoft Kinect camera sensor enables the robot to gather information about thrown objects. The Kinect is a motion sensing device that is able to gather information about the location of an object, including the depth imaging, used to calculate the distance between the camera and the object, using a speckle pattern from an infrared camera. By using the depth information, the Kinect is able to locate the object in three dimensions. By spotting object multiple times in three dimensions, it is possible to calculate the path of the moving

object, and thereby predict the impact point. This information can be sent to the robot, to enable the robot to move into position to possibly catch the object.

LEGO NXT Gyroscope

A gyroscope can be used to measure the heading of the robot. A gyroscope is constructed as a spinning disc that creates resistance when the robot is turned. This resistance is measured by the Lego NXT Gyro, and returned as a value representing the number of degrees per second of rotation.

LEGO NXT Accelerometer

An Accelerometer is a device that measures the force affecting it. The Lego NXT Accelerometer measures this information, and sends it to the robot, to provide capability for the robot to calculate its acceleration. The gyroscope and the accelerometer can in combination be used to position the robot.

Both the LEGO NXT Gyroscope and Accelerometer were used in Appendix A, but in the evaluation of the increment it was explained why these sensors are inadequate. Even though they were not used for the final product, they are cursorily explained in this section.

4.4.2 LEGO NXT Servo motor

For the robot to be able to move, it will need wheels powered by motors. In this project the LEGO NXT 9v Servo motor has been used, which at full power with no load can reach 170 RPM. The motor has a gear range of 1:48 split on the gear train in the motor. [Hurbain] The motor includes an optical fork to provide data of motor rotations within 1° precision.

For precision, and to benchmark the specific motors used in this project, a series of tests and measurements has been done on these motors:

The diameter of the wheel: 56mm
The circumference of the wheel: $56 \cdot \pi = 175.929$mm

The robot was programmed to drive forward for 10 seconds and was observed to travel a distance of 2550mm, which means it travels with a speed of 255mm/s.
The motor's RPM is: $(2230 \cdot 175.929) \cdot 6 = 86.966$ RPM. This calculation was calculated with fresh batteries, since this might affect the speed of the motors.

4.4.3 Arduino

The Arduino Mega 2560 was chosen for this project, as it has limited computational power, which will introduce interesting problems, as well as real-time constraints.

Arduino Mega 2560 Specifications:
Flash memory: 256KB (8 used by bootloader)
SRAM: 8KB
EEPROM: 4KB

Clock Speed: 16 MHz

Especially the clock speed is important, since the Arduino Mega 2560 will give 16000 clock cycles per millisecond to do the necessary tasks.

Arduino Wifi Shield

The Wifi Shield was acquired to enable wifi communication between the Kinect sensor and the Arduino Mega. The intention is to send the coordinates of the objects impact point to the Arduino, without limiting the movement of the robot. The wifi connection is able to transmit data at a rate of 9600 bits per second. [Arduino.cc, b]

Arduino Motor Shield

The Arduino Motor Shield is needed for the project to control the two DC motors independently. Without the motor shield the robot will only be able to move forward and backwards with both wheels at the same time. The motor shield needs an external power source, as the Arduino cannot provide enough power. To solve this issue, a serial circuit of two 9V batteries are attached to the shield. [Arduino.cc, a]

Design specification 5

This chapter will present the requirements and the way they are conducted for the project, the requirements will be examined at the end of the report to see if they are fulfilled. The last section in the chapter will describe the delimitations made for the project.

5.1 System requirements

The requirements for the project is assembled throughout several chapters. The analysis chapter 4, is were the most of the requirements are conducted. The general requirements are extracted from the user story in section 4.1. In the Hardware section 4.4, the hardware sets limitations for the project which lead to more detailed requirements in the form of sub-requirements.

While working on the robot, problems has occurred and the requirements therefore change. The changes done to the requirements during the project, can be seen in the increments found in the appendixes A, B and C, where the requirements for the project are in the evaluation section C.4, in appendix C.

The requirements for the project are established throughout the whole process and changes are made accordingly, to adapt to the hardware's limitations and the problems found. The requirements are listed below:

- The trash bin should catch the object if the user throws it towards the trash bin and within a predefined area
  - The robots predefined area should be calculated from the hardware limitations of the motors' speed
- The robot should know where it is positioned
  - The robot should have a starting position, from where it should be able to calculate it's current position through calculations of the motor encoders
  - The robot's starting point should be placed outside its predefined area, such that it moves forward into the area
- The robot should be able to detect and track the thrown object
  - The thrown object should be detected and tracked by a Microsoft Kinect
  - The Kinect should send the coordinates of the impact point of the trash to the robot
- The robot should be able to calculate the impact point for the object
  - Trajectory prediction should be used to calculate impact point of the thrown object

- The robot should be able to move the trash bin, such that the thrown object lands inside the bin
  - The robot should be able to turn and drive forward
  - The robot should be able to recognize the coordinates sent from the Kinect
- The robot should be able to receive data from a computer, through a wireless network
- The system tasks should be able to be scheduled and verified

5.2 Delimitations

It has been decided that for simplicity, only one object will be thrown at any given time. This delimits any multiple object tracking and prediction.
It has also been decided that the robot should not drive backwards. The robot starts on the edge of its predefined area and will only drive forward into the area. This means that if the robot is already heading towards a point in its predefined area, and the Kinect sends a new point, which is behind the robot, it has to turn 180 deg to drive to the new point given by the Kinect.

System design 6

This chapter will describe some of the software choices made, as well as the robot design. The program design, the robots positioning and movement and how the Arduino and Kinect are connected, will also be described in this chapter.

6.1 Software choices

In this section the software choices made for the Arduino IDE is described. Also, the libraries being used for making the program will be described as well.

6.1.1 Arduino IDE

While testing the Arduino IDE 1.6.12 with the code listed on the Arduino website's wifi-guide [Arduino.cc, c], it is discovered that the wifi library is not compatible in newer versions of the IDE. The library is supported in version 1.0.3, which is why this particular IDE version is used in this project. The newer IDEs have increased security, which would actively refuse any TCP connection from the computer.

6.1.2 Libraries

To make use of the Motor and Wifi Shields, it is decided that two libraries are included in the project, although similar functionality could have been written by hand instead. Constructing this functionality is not a learning goal for this project and therefore the libraries are included.

Motor.h

The Motor.h library is being used for the project, making it possible to control the motors independently of each other, e.g. the motors can be set to different speeds and can be stopped at different times.

Wifi.h

Arduino wifi library is used to create the connection between the Wifi Shield and the router. This library has methods for creating a server, where clients can both read from and write to the server. The subsection 6.1.1 explains complications between the versions of the Arduino IDE, and why the Arduino IDE 1.0.3 is used for this project.

6.2 Robot design

The purpose of the robot is to catch the thrown object, by driving to the impact point of the object and the ground within the predefined area. The final design of the robot is shown in figure 6.1.

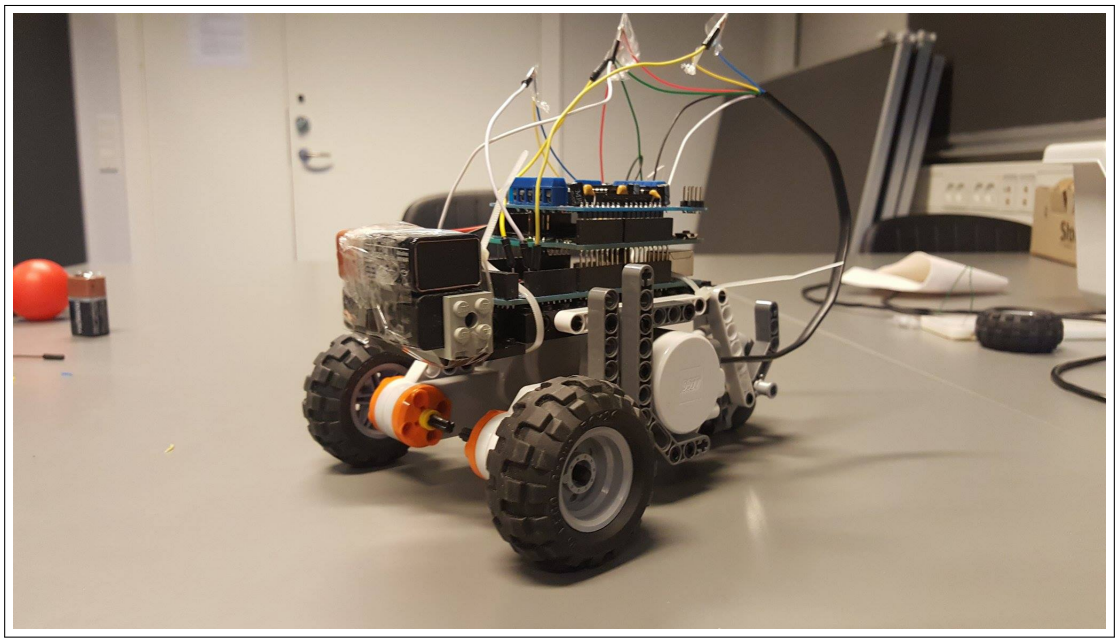

Figure 6.1. The final design of the robot

The robot uses two LEGO NXT Servo motors with compatible wheels, which will act as the front wheels. The back wheel can slide from side to side, which is important, since the wheel will be dragged sideways when the robots turns, whereas a normal rubber wheel would have caused a lot of friction. The robot is build around the NXT Servo motors, with a platform at the top. The Arduino is strapped to the platform, with the shields on top of it. This ensures that the Arduino does not slide off the robot while moving. The robot is build with the arduino at the top, for easy access to the different pins if needed.

6.3 Arduino program design

As mentioned in section 6.1, the arduino program is developed in the Arduino IDE version 1.0.3. The Arduino code controls the behaviour of the robot, which calculates the movement, keeps track of positioning, and reacts on input from the Kinect sensor. This program is responsible for the intercommunication between separate parts of the robot.

The program has two predefined Arduino functions called setup and loop. The setup function will be executed once when the program is started, and the loop function will, as the name indicates, loop over and over until the program is completed or exited. The loop function is the behaviour of the robot.
The setup function initializes the wifi connection to the computer running the Kinect software, which is used to receive predicted coordinates from the Kinect. After connecting the Arduino and the computer, the program will wait for the first set of coordinates. When

the coordinates have been received, the program will execute the behaviour according to those coordinates. The robot will move towards the coordinates, and keep track of its own position. It will check whether or not it has hit the impact point or not, and when its own position is the same as the impact point it will exit the program. The behaviour of constantly updating the position and comparing that to the impact point, and moving towards the coordinate, will loop until it has exited. The initialization and behaviour of the robot can be expressed as the following tasks:

- Make connection to the Kinect
- Wait to receive coordinates from the Kinect
- It will keep waiting for the Kinect to send a set of coordinates, when the coordinates is received, it will loop over the behaviour of the robot:
  - Check if already at the collision point, if it is exit program
  - Start moving, adjust direction relative to heading
  - Update its position

The flow of the tasks are further described with a flowgraph shown in figure 6.2.

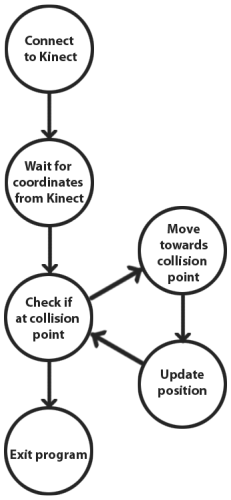

Figure 6.2. The flowgraph of the program

6.4 Robot positioning and movement

The idea for moving the robot is to send coordinates to the robot, which are converted to explicit millimetre values of the distance from the starting point (0, 0). This starting point delimits a predefined area, as described in Appendix A.3.2, which is the area in which the robot should catch the thrown object. This area is defined by the robot's capabilities and limitation. This area would allow the robot to calculate the distance travelled, and compare that value to the distance of the received coordinate. The NXT Servo motor encoders should be used to calculate the distance travelled.
If the robot has read a coordinate and started to move towards it, receiving a new coordinate would redirect the robot to the new coordinate instead. As described in Appendix A.3.4, the robot's maximum speed was 25.5 cm/s, but it is significantly slower moving backwards rather than forward. This resulted in the definition of the robot always

starting at the coordinate (0, 0), and not considering any trash thrown behind the robot. The predefined area is determined to only be in front of the robot, to ensure that the robot doesn't move backwards.

6.5 Connecting Arduino and Kinect

For convenience, and as a part of the requirements for this project, the Arduino should receive wireless data from a computer, connected to the Kinect. The Arduino Wifi Shield makes this possible, with the Wifi library included in the Arduino program.
The connection between the Arduino and the computer should be through a local router, with a set SSID and password. The computer should use the Kinect to calculate an impact point of the object, and send this in a specific format, so that the data can be easily read and translated to a set of coordinates for the robot's movement. The computer should be able to send coordinates more than once, since the first impact point is not necessarily precise enough to catch the object. The computer should calculate increasingly accurate impact points, which should be sent with a certain minimal inter-arrival time, to not hinder the robot's movement, and yet still in time for the robot to correct itself.
The Arduino should receive this data and read whenever it has sufficient time to do so, and convert the data received to match the right coordinate in the predefined area.

6.6 Scheduling

The subsection, Cyclic Executive, details the scheduling model used in this project. The tasks, and the model type, will be explained, and further analysed in the subsection, Real-Time analysis, to express information about how the scheduling was achieved. The subsections, Interrupts and their definitions, Stack Overflow and Interrupt Overload use the article "Safe and Structured Use of Interrupts in Real-Time and Embedded Software" written by John Regehr [Regehr, 2006].

6.6.1 Cyclic Executive

A cyclic executive is a model which assumes a fixed set of periodic tasks. The model is about designing an entirely static schedule in which the tasks are cyclically executed at their rate, since they are periodic, and must also meet their deadline. The model is cyclic since when the last task ends and the allotted time of a cycle ends, a new cycle begins, with all tasks placed in the same specific order as the previous cycle. The creation of such a schedule proves by construction that the tasks will always meet their deadlines at runtime, if the assumptions of the model are true.
To ensure that the cyclic executive is in synchronization with real elapsed time, synchronization points are inserted into the code that implements it. Once cyclic executives are constructed, the implementation is simple and efficient since there is little overhead as there is no scheduling at runtime. Cyclic executives can be constructed separately from the system, by hand or by a tool. [Bertolotti og Manduchi, 2012]

6.6.2 Interrupts and their definition

When describing interrupts, one must first define what interrupts are. The definitions are quotes from John Regehr's article. The definition for an interrupt is twofold, as the first part is as follows:

> "A hardware-supported asynchronous transfer of control to an interrupt vector based on the signaling of some condition external to the processor core"

The second part of the definition is:

> "An interrupt is the execution of an interrupt handler: code that is reachable from an interrupt vector"

The definition for the introduced term interrupt vector:

> "A dedicated or configurable location in memory that specifies the address to which execution should jump when an interrupt occurs"

Interrupts don't always return control flow to where the interrupt disrupted control flow. Interrupts often change the state of main memory, and that of device drivers, but does not disturb the main processor context of the computation which was disturbed.
An interrupt is pending when it's firing condition has occurred, the interrupt controller has been updated and the interrupt handler has not started executing. A missed interrupt is when the firing condition occurs but it does not become pending, this is usually because another interrupt of the same type is already pending. Most hardware platforms, including the atmega2560, use a bit to distinguish whether an interrupt is pending or not, which means it does not track if there is more than one interrupt pending of that type. Hardware support can be used to disable interrupts by manipulating bits in hardware registers, either through the master interrupt enable bit, or the enable bits for each interrupt.

The following conditions decide when the firing condition for an interrupt is true:

- The interrupt is pending
- The processors master interrupt enable bit is set
- The enable bit for each interrupt bit is set
- The processor is in between executing instructions
- There exists no higher priority interrupt which fulfills 1-4

Another important part of interrupts is interrupt latency, which is the time between the interrupt firing conditions become true, and the first instruction of the interrupt handler has begun executing. Nested interrupts is when an interrupt handler is preempted by another interrupt. An important distinction between threads and interrupts is that thread scheduling is through software, while interrupt scheduling is through hardware interrupt controller.

6.6.3 Stack Overflow

A stack overflow is when a stack grows beyond the confines of the memory allocated to it, thereby corrupting RAM and could cause system malfunction and crash. Allocating memory to a stack is about a balance between enough memory to prevent stack overflow

from occurring and not assigning more memory than needed. There are two ways to approach this problem, analysis and testing based. A significant advantage of analysis is that it can be performed quickly through tools, in contrast to testing which is much more laborious.
The testing approach is about running the system and observing how large the stack grows. This approach is a kind of black box testing, it doesn't matter how or why memory is used, all that matters is how big the stacks becomes. A problem with the testing based approach is that it can miss some of the program paths, where it might have taken a branch at some point during the execution of the program which causes greater stack growth than the paths tested.
The analysis based approach is concerned with the control flow and its goal is to find the path which pushes the most data onto the stack. The problem with the analysis based approach is that it's often too optimistic about the maximum stack size to prevent stack overflow, since it accomadates behaviour which occur very rarely or never.
Stack overflow is a general problem in embedded systems, but was not tested for this project, this is addressed as part of 11.

6.6.4 Interrupt overload

Interrupt overload is when interrupts occur so frequently they dominate the processor from performing its other computations, which in the worst case can end up starving other important processes. While it may be intuitive to think that large interrupt payloads are a natural cause of interrupt overload, but it is rather unexpectedly large interrupt loads which can cause interrupt overload. A reliable maximum interrupt request rate is a important thing to ensure that it is accurately evaluated to have a reliable real-time system.

To discover if interrupt overload can occur, it is possible to analyze how much cpu utilization can be spent handling interrupts, the maximum time spent in an interrupt handler can be calculated by the maximum execution time of a given interrupt multiplied by the worst-case arrival rate.
The maximum execution time of the interrupt handler in our system is 40 clock cycles, 10 of which are used to enter and exit the interrupt handler with the remaing 30 being the actual interrupt handler code, and the worst-case arrival rate is 1 occurrence per millisecond. The worst-case arrival rate is based on the maximum speed of the NXT servo motor and the motor encoder which reads it, which causes interrupts to be generated.
The maximum time spent in an interrupt handler in our system is $\frac{40 \cdot 1}{16000} = 0.0025\%$ of cpu spent dealing with interrupts, where the 16.000 comes from the cpu clock cycles per millisecond. The interrupts in this project listens to the motor encoders for change, each time a change happens(the change is between high and low voltage), it will interrupt the program and in this project it will increment a counter. Since only one type of interrupt with a relatively low amount of cpu utilization is used, interrupt overload cannot occur for this system.

Real-Time analysis

The previously addressed problems of stack overflow and interrupt overload are subproblems of the real-time schedulability analysis, especially concerning interrupts, which will make sure that all computations will be met within their time constraints.
To schedule the system a cyclic executive is used, which is mentioned in section 6.6.1, which handles many of the problems addressed, as long as its assumptions are true. Since the cyclic executive has predetermined behaviour it will have no jitter for the execution of each of the periodic tasks. Although jitter has not been a particular concern in this project.
Jitter is the concept of time variation in a computed output being messaged to external environment from period to period.
The strengths of using a cyclic executive include: execution schedule is predetermined, simple, efficient and fast, no jitter, prevents race-conditions and deadlock. The existence of a cyclic executive is a proof by construction that the real-time analysis will hold, and the problems which arise from interrupts have also been addressed in this chapter.

Implementation 7

This chapter will describe the implementation of the systems components. It will cover the description of the Microsoft Kinect, the robot's code and how the scheduling is done.

7.1 Microsoft Kinect

This section will describe how the Kinect sends coordinates to the Arduino. The Kinect gives information about the object predicted impact point, in relation to its field of view and the distance to the point. The coordinate has to be converted to fit in a coordinate system that has it's starting point at the Kinect, X-axis out from the side of the Kinect, and Y-axis out perpendicular from the front of the Kinect.
A detailed description of the conversion can be found in C.3.1.

The coordinate can then be sent to the Arduino, using its IP-address and a tcpClient. When writing to the tcpClient's stream, the information is sent to the server on the Arduino, that should be waiting to retrieve it. The code for this can be seen here:

Listing 7.1. Sending data to the Arduino

```
1 tcpClient = new TcpClient();
2 tcpClient.Connect("192.168.0.100", 9999);
3 stream = tcpClient.GetStream();
4
5 stream.Write(Encoding.UTF8.GetBytes((finalx.ToString() + ";" + finaly.ToString()).ToCharArray()),
      0, (finalx.ToString() + ";" + finaly.ToString()).Length);
6 tcpClient.Close();
```

7.2 Robot

This section will describe arduino code for the robot. This will include how the arduino connects to the Kinect and the robots behaviour and calculations done by the arduino. If there was any comments in the original code, they have been removed in the listings for this chapter.
A problem occurred when both the Wifi Shield and the Motor Shield should work together. When the Wifi Shield is trying to connect to the network and if any of the Motor Shield components are initialized at the beginning of the program, it isn't possible to connect to the network. If it is connected to the network before the Motor Shield components are initialized, it is possible to receive data and use the Motor Shield components.

The arduino setup function, is a function executed once at the beginning of the running program, to setup the program as the name implies. In listing 7.2 the setup function is shown, which in this project will connect to the private network, using the guide from

Arduino's own website, and get the first coordinate [Arduino.cc, c].
It will try to connect to the specific network by passing the network name and password to the function WiFi.begin(), given by the Wifi library. If the Arduino can't connect to the network, the program will exit. If it successfully connects, the arduino will call the function server.begin() to initialize a server that can receive data, and is now connected to the network.
The **dataString** variable contains the data received, so the while loop makes sure that the program waits for the coordinates before continuing. A client variable will be initialized using server.available(), which will make the Arduino ready to receive data. The function client.available(), returns true if the server has received any information. If true, the data is appended to the dataString, and execution of the program will continue.
goalX and goalY is then extracted from the **dataString** and converted to integers.

Listing 7.2. The arduino setup function

```
1  void setup() {
2         Serial.begin(9600);
3         status = WiFi.begin(ssid, pass);
4         if ( status != WL_CONNECTED) {
5                 exit(0);
6         }
7         else {
8                 server.begin();
9         }
10        ip = WiFi.localIP();
11
12        while(dataString == ""){
13                client = server.available();
14                if (client.connected()){
15                        while (client.available()) {
16                                char c = client.read();
17                                dataString += c;
18                        }
19                }
20        }
21        goalX = dataString.substring(0,dataString.indexOf(';')).toInt();
22        goalY = dataString.substring(dataString.indexOf(';')+1).toInt();
23
24        pinMode(LEFTENCODERPIN, INPUT);
25        attachInterrupt(3, incrementLeft, CHANGE);
26        pinMode(RIGHTENCODERPIN, INPUT);
27        attachInterrupt(2, incrementRight, CHANGE);
28
29        leftTemp = leftTotal;
30        rightTemp = rightTotal;
31 }
```

Now that the robot knows its destination, it needs to calculate how to get there, but before it does that it needs to know its own position and heading. The system used to determine position and heading is defined as follows:

Position is defined as a x and y coordinate like so: (x,y).
Heading 0 is straight up parallel to the y axis
Headings clockwise from the y axis are negative and headings counterclockwise are positive.
Heading can range from -179deg to 180deg.

The robots motors each have an encoder that will give a high volt signal to the Arduino for every 2 degrees the motor turns. The Arduino is set up to interrupt on pin change, which means the interrupt handler for each motor will be executed for each degree the motor turns. Each interrupt handler increments an int, leftTotal for the left motor and rightTotal for the right motor.
Using these two variables, position and heading of the robot can be calculated, or rather the delta position and heading compared to the last time position and heading was calculated. First the values of leftTotal and rightTotal are saved to two new variables currentLeft and currentRight such that they do not change while using them in calculation. To record how far the robot has moved since the last calculation, the last iterations degree count called temp(Left/Right) is subtracted from the current to get the delta degree count. To get the actual distance each wheel has moved, the delta degree count is multiplied by the amount of millimeters the wheel travels per degree. The position of the robot is set to be between the wheels, and to get how far the position has moved the two distances that the wheels have moved are added together and divided by 2. Next, to find the new position in coordinates the distance traveled is multiplied with sinus to the heading to get the change in the x direction and cosinus to the heading to get the change in the y direction. These two values are added to the current position to get the new position. To get the change in heading, the distance the right wheel traveled is subtracted by the distance the left wheel traveled to get the distance the two wheels traveled compared to each other. Arctan to this value divided by the distance between the wheels gives the change in heading, which is added to the current heading to get the new one. This is shown in listing 7.3

Listing 7.3. updatePosAndHead function

```
1  void updatePosAndHead(){
2          int currentLeft = leftTotal;
3          int currentRight = rightTotal;
4          double deltaLeft = (currentLeft - leftTemp) * DISTPRDEGREE;
5          double deltaRight = (currentRight - rightTemp) * DISTPRDEGREE;
6          leftTemp = currentLeft;
7          rightTemp = currentRight;
8          double dist = (deltaLeft + deltaRight) / 2.0;
9          posX += (dist * sin(heading));
10         posY += (dist * cos(heading));
11         heading += (atan((deltaRight - deltaLeft) / WHEELDIST));
12 }
```

The robot now knows its destination, its own position and heading. To get to the destination, the robot can do 4 things: drive forward on one wheel and let the other stand still or vice versa, drive both wheels forward or stand still on both wheels. So basically left, right, forward or nothing. The Arduino first determines if the robot is already at the goal position or it needs to move. To do this it simply compares its current position with the final destination and determines if this difference is within a given margin of error. If it is, it stops and exits the program, this is shown in listing 7.4

Listing 7.4. If-statement to check if the robot should stop

```
1  if(posX <= goalX + margin && posX >= goalX - margin && posY <= goalY + margin && posY >= goalY -
       margin){
2          motorLeft.run(RELEASE);
3          motorRight.run(RELEASE);
4          delay(50);
5          exit(0);
6  }
```

If it is not at the goal position, the robot needs to be moved. To figure out whether to go left, right or forward the arduino calculates the goal heading that is needed at the current position to point directly (or almost directly) at the goal destination. Next this heading is compared to the current heading and the arduino determines whether a right, left or no turn is needed. This is done like this: First the goal position is subtracted by the current position to get how far the robot needs to travel in the x and y direction, from here on called deltaX and deltaY.
If deltaX is 0, the goal is either straight up parallel to the y axis at heading 0 or straight down parallel to the y axis at heading 180 or PI in radians.
If deltaX is bigger than 0, arctan is taken to deltaY divided by deltaX, this gives the angle to the positive x axis. This is subtracted by 90 degrees (PI/2 in radians) to get the angle to the positive y axis, which is the goal heading.
If deltaX is smaller than 0, almost the same thing as above happens. The only difference is that the angle from the arctan calculation will give en angle to the negative x axis, so the results is added to 90 degrees (PI/2 in radians) instead of subtracted by it. Shown in listing 7.5.

Listing 7.5. Part of the driveTowardsGoal function to calculate the goalHeading

```
1 double deltaX = goalX - posX;
2 double deltaY = goalY - posY;
3 double actualGoalHeading;
4 if(deltaX == 0 && deltaY > 0)
5         actualGoalHeading = 0;
6 else if(deltaX == 0 && deltaY < 0)
7         actualGoalHeading = PI;
8 else
9         actualGoalHeading = deltaX >= 0 ? atan(deltaY/deltaX) - (PI/2) : atan(deltaY/deltaX) +
              (PI/2);
```

The difference between the current heading and goal heading is then calculated by subctracting the current heading from the goal heading, this is from here on called the deltaHeading.
If deltaHeading is larger than 180 degrees (PI in radians) it would be faster to go the other way around, so deltaHeading is subtracted by 360 degrees (2PI in radians).
if deltaHeading is smaller than -180 degrees (-PI in radians) the same thing aplies, but the other way so 360 degrees (2PI in radians) is added instead.
Lastly it is determined if the deltaHeading calls for a left turn, a right turn or full speed ahead, as shown in listing 7.6.

Listing 7.6. Check the difference of the actualGoalHeading and its current heading, and determines which motors to run

```
1  double deltaHeading = actualGoalHeading - heading;
2  if(deltaHeading > PI)
3         deltaHeading -= 2*PI;
4  else if(deltaHeading < -PI)
5         deltaHeading += 2*PI;
6  if(deltaHeading < -0.1){
7         motorLeft.run(FORWARD);
8         motorRight.run(RELEASE);
9  }else if(deltaHeading > 0.1){
10        motorLeft.run(RELEASE);
11        motorRight.run(FORWARD);
12 }else{
13        motorLeft.run(FORWARD);
```

```
14          motorRight.run(FORWARD);
15  }
```

In the loop, a dummy function has been put in to simulate the wifi part of the code, which should run every iteration, but because of shield incompatibility it cannot. Getting a string of coordinates from the sender takes less than 9 ms C.2.2, so the dummy function is simply a delay of 9 ms. After this updatePosAndHead() and driveTowardsGoal() are each executed once. Next a delay is inserted to let the robot drive briefly before calculating the change in position again. After the delay updatePosAndHead() and driveTowardsGoal() are run a second time, and then a while loop waits for the time since starting the loop to exceed 33ms. This is done because the expected minimum delay between receiving information from the kinect is 33ms as the framerate of the kinect is 30 frames pr. second. updatePosAndHead() and driveTowardsGoal() are run twice to make sure the difference between updates isn't too large. If the differences are to big the robot will get unprecise position and headings after a while.

Listing 7.7. Loop function

```
1
2   void loop() {
3           t = millis();
4           wifi();
5           updatePosAndHead();
6           driveTowardsGoal();
7           delay(10);
8           updatePosAndHead();
9           driveTowardsGoal();
10          while(micros()-t<33){}
11  }
12
13  void wifi(){
14          delay(9);
15  }
```

7.3 Scheduling

To be able to schedule the systems functions(also called tasks), the worst-case execution time(WCET) have to be known for the individual functions. The first attempt was to count the clock cycles in the assembly file of the complied arduino code. It was known that the Arduino mega 2560 has 16000 clock cycles per millisecond, so it will be possible to calculate the time for the functions. The purpose was to count the clock cycles for the two functions driveTowardsGoal() and updatePosAndHead(), to make that a possibility many of the functions in the different libraries used also had to be counted, the results of this can be found in Appendix D.

In the following list, the clock cycles counted of the two functions are shown:

- driveTowardsGoal = **4185** + (13)* + (174 + (13)*)* + (231 + (13)*)* + (17)* + (8)* + (33 + (17)*)* + (7)* + (5)* + (290 + (5)*)*
- updatePosAndHead = **3560** + (11)* + (13)* + (7)* + (174 + (13)*)* + (231 + (13)*)* (10)* + (24)* + (17)* + (33 + (17)*)*

The bold number is the worst-case of clock cycles counted in the function. The numbers in parentheses are loops, which bound is unknown, therefore it is not possible to make a count the exact WCET by counting the clock cycles.

Since it was not possible to count the clock cycles from the assembly code, it was decided to use the tool Bound-T, which will calculate the WCET outputted with an output in clock cycles.
Bound-T was not able to calculate the WCET for the code. When Bound-T gets to calculating the floats, it is failing. It can't set an upper bound for the floating point numbers' WCET.

To be able to work around the float issues with Bound-T, a library called AVRFIX made by Maximilian Rosenblattl and Andreas Wolf [Rosenblattl og Wolf]. In the listings 7.8 and 7.9 a function written with the AVRFIX library and a function written normally for arduino can be compared. Be aware that the function in listing 7.9 might not look exactly like this in the final code for the project.

Listing 7.8. The function updatePosAndHead with AWRFIX library

```
1  void updatePosAndHead(){
2          int currentLeft = leftTotal;
3          int currentRight = rightTotal;
4          fix_t distPrDeg = ftok(DISTPRDEGREE);
5          fix_t dltL = itok(currentLeft - leftTemp);
6          fix_t dltR = itok(currentRight - rightTemp);
7          fix_t deltaLeft = mulk(dltL, distPrDeg);
8          fix_t deltaRight = mulk(dltR, distPrDeg);
9          leftTemp = currentLeft;
10         rightTemp = currentRight;
11         fix_t deltaSum = deltaLeft + deltaRight;
12         fix_t dist = divk(deltaSum,ftok(2.0));
13         fix_t sinHeading = sink(heading);
14         posX += mulk(dist, sinHeading);
15         fix_t cosHeading = cosk(heading);
16         posY += mulk(dist, cosHeading);
17         fix_t rel = divk((deltaRight - deltaLeft),ftok(WHEELDIST));
18         heading += atank(rel);
19 }
```

Listing 7.9. The function updatePosAndHead from the Arduino IDE

```
1  void updatePosAndHead(){
2          int currentLeft = leftTotal;
3          int currentRight = rightTotal;
4          double deltaLeft = (currentLeft - leftTemp) * DISTPRDEGREE;
5          double deltaRight = (currentRight - rightTemp) * DISTPRDEGREE;
6          leftTemp = currentLeft;
7          rightTemp = currentRight;
8          double dist = (deltaLeft + deltaRight) / 2.0;
9          posX += (dist * sin(heading));
10         posY += (dist * cos(heading));
11         heading += (atan((deltaRight - deltaLeft) / WHEELDIST));
12 }
```

The program was rewritten with the AVRFIX library and then using Bound-T again to calculate the worst-case execution time for the functions. The AVRFIX library came with a .txt file from the GitHub download, which included the clock cycles for the AVRFIX functions.
The function updatePosAndHead() is calculated to use 3995 clock cycles, but when trying

to calculate the function driveTowardsGoal() another problem occurred. When a division by 0 took place, the program will enter an infinite loop, making it impossible to set an upper bound on the WCET. With this problem it is not possible to calculate the WCET for driveTowardsPoint().

Because using the AVRFIX library with Bound-T also failed, the group decided to use the function micros() on the functions to measure the time spent on the function, run them several times and use highest result as WCET. Besides the functions updatePosAndHead() and dirveTowardsGoal() the WCET, when the Kinect program sends data till the arduino received it via the WiFi, was also measured.

The WCET for the two functions and the time spent on sending data from the Kinect via wifi, using the Micros() function, is listed below:

- updatePosAndHead = 1076 microseconds
- driveTowardsGoal = 732 microseconds
- WiFi = 8433 microseconds

These results will be used to make a schedulability analysis in the UPPAAL program, which will be described in the following chapter.

Tests 8

The following chapter will describe the schedulability analysis made with UPPAAL, the unit tests and a functionality test made for the solution.

8.1 UPPAAL schedulability analysis

As described in appendix C section C.3.2, Dumpsty contains four tasks: PrA, PrB, PrC and PrD. These three tasks are instances of the three tasks expressed in section 7.3, PrA being updatePosAndHead, PrB is driveTowardsGoal, PrC is WiFi and PrD is the delay to not run all tasks seven times within the deadline. The tasks all have individual WCET, which is not calculated, but rather tested, since calculating these through assembly code proved to be impossible due to unbound loops in libraries. After testing the individual tasks WCET, the worst case found would be significantly faster than what is labelled in UPPAAL, since the probability of hitting the actual worst case is close to impossible with the amount of tests done. In the following bulletpoints, all three tasks tested WCET and the WCET used in UPPAAL for the specific task is expressed, which is the first step to verify the schedulability of Dumpsty's tasks.

- PrA Tested: 1076 microseconds UPPAAL: 2 milliseconds
- PrB Tested: 732 microseconds UPPAAL: 1 milliseconds
- PrC Tested: 8433 microseconds UPPAAL: 9 milliseconds
- PrD Tested: 10000 microseconds UPPAAL: 10 milliseconds

For convenience, and to simplify the analysis, all tasks contain the worst case runtime of all interrupts and interrupt handlers that might occur during the execution of the task. These interrupts are generated by the motor encoders when Dumpsty is moving.

Figure 8.1 depicts the automata created in UPPAAL from two declared classes. The first class, task PrA, is the cyclic executive instance for the task PrA. The second class is simply a CPU which is the key needed to run the task. Every task can grab the CPU, but only one may hold it at any given time. The CPU is then released when the task is done executing, and another task can then proceed to run.

In UPPAAL these automata might have more than one instance, defined by the amount of tasks declared. In this case, there are three instances of the first class:

- PrA = TASK(1, 33, 2);
- PrB = TASK(2, 33, 1);
- PrC = TASK(3, 33, 9);
- PrD = TASK(4, 33, 10);

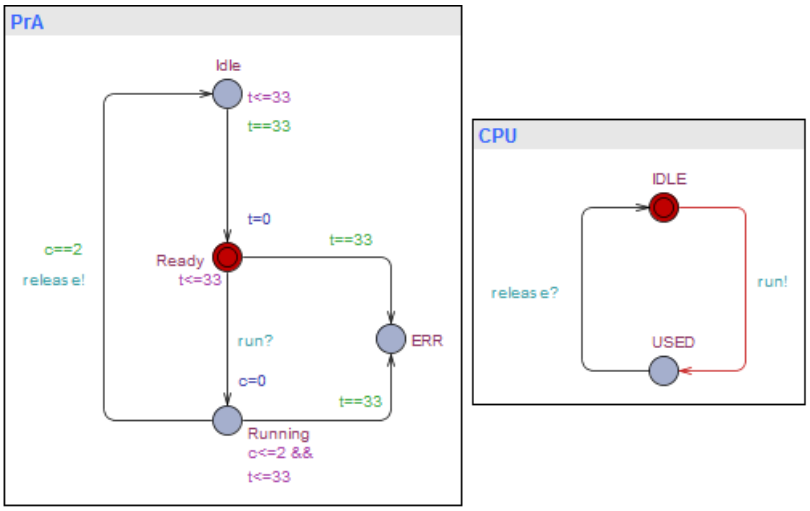

Figure 8.1. Automata in UPPAAL

The integers declared for every task have different meanings. These integers will be explained by the case task PrA. In task PrA, the integer 1 is the ID for the task, 33 is the deadline for the task in milliseconds, and 2 is the worst-case execution time for the task. The deadline for all tasks are the same, since this is the minimal interarrival-time(MIT) for coordinates from the Kinect sensor. This MIT is granted since the Kinect has a maximum FPS of 30, and each new frame can trigger a new prediction. There cannot be a new prediction without new frames, as the frames hold all the data for the Kinect prediction. All code has to be executed within the MIT of these coordinates, since a new coordinate will alter the path the robot choose.

The last step in the UPPAAL schedulability analysis is to use the verifier. This is done with two queries, found in listing 8.1. The first query checks if, after some amount of time, greater than zero, all tasks has been executed at least once and all tasks are in the ready state. The second query checks that no task ever hits an error-state. If these two queries both succeed, the tasks can all be executed within the deadline, and the schedulability analysis is successful, and in this case, with tasks PrA, PrB and PrC, the tasks are schedulable within the deadlines.

Listing 8.1. Queries for UPPAAL

```
1 E<> PrA.Ready and PrB.Ready and PrC.Ready and PrA.t==0 and PrB.t==0 and PrC.t==0 and time>0
2 E[] not (PrA.ERR or PrB.ERR or PrC.ERR)
```

8.2 Unit test

To make the unit test on the arduino code, the library ArduinoUnit has been used [MacEvoy et al.]. For the arduino code there has been conducted two different types of unit tests, testing the two functions driveTowardsGoal() and updatePosAndHead(). Both functions has six different cases of unit tests. Each unit test has a name expressing the different variables given in the test, in listing 8.2 the unit test has the name curPos30_150goalPos30_150Head1 since the current position of the robot is (30, 150), the goal position is (30, 150) and the current heading of the robot is 1.
The coordinates sent to Dumpsty in the unit tests are simulated, since this is the best

representation of how the robot would work under perfect conditions. These unit tests reflect how Dumpsty performs in an environment with instantaneous received data from a sensor with great precision, where the sensor will send new data every 33ms. The first type will test the function driveTowardsGoal() and has 3 different inputs being starting position, goal position and the heading of the robot. The robot will start at its starting position with the given heading, and will then have to move towards the goal position. In the description of the test, if no heading is described, the heading is 0. For a code near perspective of this unit test type, check listing 8.2.

Listing 8.2. First type of Unit test

```
1 test(curPos30\_150goalPos30\_150Head1){
2 heading = itok(1);
3 posX = itok(30);
4 posY = itok(150);
5 goalX = itok(30);
6 goalY = itok(150);
7 String result = driveTowardsGoal();
8 assertEqual(result, "goal");
9 }
```

The second type of unit test will test the function updatePosAndHead(). The test takes 4 inputs being the starting position, the heading, millimetres to move on the right motor and millimetres to move on the left motor. The robot will start at its starting position with the given heading, and will move the given millimetres for each motor. Then the unit test will check if the robot is within 0.01mm of the asserted position. An example of this type of unit test can been found in listing 8.3.

Listing 8.3. Second type of Unit test

```
1  test(pos300_250Head075Left12Right5){
2  heading = ftok(0.75);
3  posX = itok(300);
4  posY = itok(250);
5  rightTotal = 5;
6  rightTemp = 0;
7  leftTotal = 12;
8  leftTemp = 0;
9
10
11 updatePosAndHead();
12
13
14 assertMore(posX, ftok(302.826));
15 assertLess(posX, ftok(302.836));
16
17
18 assertMore(posY, ftok(253.034));
19 assertLess(posY, ftok(253.044));
20
21
22 assertMore(heading, ftok(0.717));
23 assertLess(heading, ftok(0.727));
24 }
```

Table 8.1 includes all the unit tests made for the arduino code. The table consists of a description of the unit test and if the test passed or failed.

Test description	Result
The robot is at position (30, 150), and gets the same coordinate sent, this will check if the robot says it is at goal or will try move to a new position.	Passed
The robot will have a starting position at (0, 0) and will be given a new coordinate at (0, 150), the robot should then drive forward in a straight line.	Failed
The robot will start at position (0, 0) with a heading of (1) and will be given the new coordinate (0, -150) which should make the robot turn left.	Passed
The robots starting position is (150, 150) with a heading of (-1) and is given a new coordinate at (140, 100). This means the robot will be pointing towards the top right and will have to drive down towards the right.	Failed
The robot starts in position (150, 150) with a heading of (-1) making the robot point towards the top right. The robot is then given a new coordinate (300, 200), the robot should then turn around it self and move towards the top left	Passed
The robots starting position is (0, 0) with a heading of (1), making the robot point towards the top left. The robot is given a new coordiante (0, 300). The robot will only use one motor till its heading is 0, and then drive forwards to the new coordinate	Passed
The robots starting position is (0, 0) and is given the new coordinate (-150, 300). The robot should the move towards the new position at the top right.	Passed
The robot has the starting position (300, 250) with a heading of (0.75). The robot is ordered to drive 5mm with the right motor, and 12mm with the left motor and check if the new position is within a precision of 0.01mm of the asserted position.	Passed
The robot has the starting position (160, 100) with a heading of (-0.4) and is to drive 14mm with the right motor and 3mm with the left motor. The robot should be within 0.01mm of the asserted position	Passed
The robot is at starting position (120, 200) with a heading of (1). The robot is to drive 10mm with each motor, and be within 0.01mm of the asserted position	Passed
The robot has the starting position (0, 0) and the robot is to drive 10mm with the right motor. The robot should be within 0.01mm of thte asserted position	Passed
The robot has the starting position (0, 0) and the robot is to drive 10mm with the left motor. The robot should be within 0.01mm of thte asserted position	Passed

Table 8.1. Table of conducted unit tests

All the failed test have been corrected and is now working as expected.

8.3 Functional test

Now that the unit tests for Dumpsty are successful, the functionality of Dumpsty must be tested as well. This is done through a functional test, where Dumpsty is placed in an appropriate environment, and then the entire system will be tested. This is done through throwing an object in front of the Kinect sensor, having the Kinect print the sent coordinates, and having the robot moving to the point. The actual impact point of the object will also be marked on the ground, and the distance between all three points will be measured in cm, shown in table 8.2.
The three columns of table 8.2 depicts the following:
Predicted to impact: The distance from the predicted coordinate from the Kinect, to the actual impact point of the object.
Predicted to robot: The distance from the predicted coordinate to the robot, after the robot has stopped moving.
Robot to impact: The distance from the robot after it has stopped moving, to the actual impact point of the object.

Predicted to impact (cm)	**Predicted to robot (cm)**	**Robot to impact (cm)**
23	11	34
11.5	6.5	14
19.5	8	26
19	10.5	26.5
36	8.5	43.5
48	11	56
9.5	11	21
16	5.5	11
49.5	11	48.5
13	11	19

Table 8.2. Table of conducted functional tests

8.4 Summary

The scheduling is rather simple since the tasks are small and the platform to perform the tasks is faster and has more memory than needed, as shown in the schedulability analysis. The unit tests are implying that the robot should be working as intended, since it passed most of the tests and the tests which failed were corrected. The functionality test shows that the Kinect didn't correctly predict where the impact point of the object is, and the robot doesn't stop at the exact coordinate point sent by the Kinect.

As one can see, Dumpsty is consistently close to 10 cm from the predicted coordinates, which could be caused by the servo motors drift after the power to the motors has been shut off. The fact that the prediction has a range of misprediction of 10 - 50 cm is either due to miscalculations in the Kinect code, or the Kinect having hardware issues, creating a distorted depth-picture.

Discussion 9

The discussion chapter will describe some of the choices made through the project and problems that have occurred. The problem and choices will be described and discussed, what happened and what could have been done differently.

Scheduling

For the schedulability analysis, the WCET of the Arduino code had to be calculated. In the course, we learned that this could be done either with a language constructed timer, by counting clock cycles in the assembly code, or by using a tool. In our case, we used the tool Bound-T, without success. Our first idea was that a tool could provide WCET calculations for a piece of code, but because we were using libraries and loops which are not supported by Bound-T, it reported errors without a usable output. So our next idea was to calculate the WCET by analyzing the assembly code, but yet again the libraries proved to be a problem, since the source code of some libraries were inaccessible and contained unbounded loops. We tried counting the clock cycles, as shown in Appendix C.2.2. The next much coarser solution we tried was to estimate a worst-case running time for entire pieces of code, by using the timer implemented in the Arduino IDE. This is not a provable WCET estimate, since the odds of ever hitting the worst run time would be close to nonexistent in the little sample-size we provided. This lead to us almost doubling the WCET estimate in the UPPAAL model, as we already knew that the tasks were schedulable, even with a far greater WCET than what was estimated by the timer.

Hardware choices

To introduce other classical real-time concerns such as memory and execution time into this project, another platform smaller and slower than the Arduino Mega 2560 should've been chosen, since we only partly utilized the Arduino Mega 2560's processor power.

The two shields and their libraries for this project, the Arduino Motor Shield and the Arduino WiFi Shield, are helping us writing code to control the NXT servo motors and connecting to the network. When using both shields at the same time, some complications occurred. This is because the shields are using some of the same pins on the Arduino board. To work around this problem, it was decided to change the Arduino code such that it first connect to the network which will change a boolean to true and then initialize the motor shield components for use. This also means that it is not possible to continuously get new data from the Kinect, as it will send the first coordinate it captures and predicts of the thrown object.

The motor shield should be used to control the motors, but an other solution to send the Kinect data to the arduino shield could be used. Either we should look for another WiFi component for the arduino, a bluetooth component or the robot should just be connected to the computer controlling the Kinect via a USB cable.

As mentioned in chapter 8 about tests, the Kinect doesn't reliably predict where the thrown object lands. In this project the depth sensor of the Kinect has been used, to calculate the impact point of the thrown object. Since the Kinect didn't predict the impact point reliably, either the calculations made in the code should be revised or another way of using the Kinect should be taken into consideration. It is also possible that the Kinect didn't function properly.

Another important hardware issue in this project is the NXT servo motors. These motors capabilities severely limits the whole system's capabilities. The motors set the boundaries of how far Dumpsty can move within a certain time limit paired with the circumference of the wheels. Having faster motors would also help on the response time of Dumpsty.

Delimitations

For this project we made two delimitations to the project, which can be found in section 5.2. As mentioned, only one object will be thrown at any given time. This is an environmental delimitation, to not consider multiple objects at once.
Another delimitation made was that the robot should not drive to coordinate points behind itself within the predefined area, but this is actually possible for the robot to do, also outside the predefined area. A problem we should have realized much earlier is that it is not possible for the robot to catch the object, unless it lands extremely close to the robot's starting position, because of time spent processing, sending a signal and limitations of the motor. Another delimitations should have been made that the robot should not try to catch the object, but rather drive to the point where the object hit the ground.
Instead the predefined area should not be considered and the robot should have a starting point at (0, 0), as shown in figure 9.1 as the red dot. The robot should then be able to move to both of the blue crosses(direction of the old predefined area, but not limited to the old predefined area) and the green crosses behind the robot's starting position. Meaning the robot only have a starting position at (0, 0), and the distance it can move on either axis is not limited, as it was with the predefined area. The robot should then drive to the predicted impact point of the object and just mark where the object had landed.

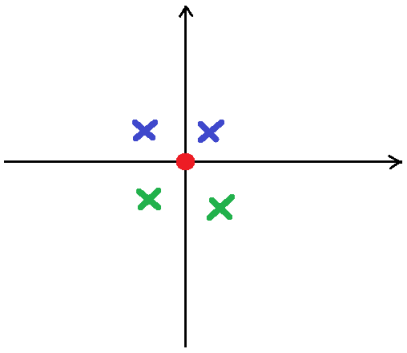

Figure 9.1. Graph describing the coordinate set with no predefined area

Process analysis

The process model used for this project is a reflective model of the processes we have constructed our earlier projects from. This process model is based on our experience with problem based learning, and includes the processes we usually undergo through a project. It is inspired by XP, including pair programming and agile development with incrementally changing requirements. If the solution to our problem would be a safety critical solution, a more test-driven process model would be more suitable.
The process model itself has worked out rather well since it isn't restrictive and provides a very large degree of freedom when working with the project, compared to a rigid adherence to a well established model, such as the waterfall model.
After experimenting with the process model throughout this project it has come to our attention that some changes should be made to the structure of the process model. A more explicit test section should be implemented to test all interdependent components, and ensure that all increments' implementations work as intended. This will in turn also increase the coherence between each increment. This test section will also include tests across each increment, to check that no new errors has been introduced to earlier implementation.

Conclusion 10

Based on chapter 4 concerning a user story, system capabilities, hardware's capabilities and limitations, lead to the project's problem statement:

How can an embedded system control a trash bin to detect, track and catch a thrown object within a designated area?

The final system partially solves the problem statement. It is capable of detecting and tracking the object thrown towards the robot, but not consistently catch it, because of the hardware limitations for the project, these limitations are mentioned in the discussion 9.

To solve the problem statement, a list of general requirements with related sub requirements were constructed. In the following paragraphs each of the general requirements will be evaluated.

The trash bin should catch the object if the user throws it towards the trash bin and within a predefined area

This requirement was not accomplished, since the robot was not reliably able to move to the impact point of the thrown object. No trash bin has been attached on top of the robot, which would determine the area in which it is able to catch the object, but it is still considerably inaccurate by 10-50 cm from the actual impact point to the robot's final position. If the estimated coordinate sent by the Kinect was true, it would consistently be 11 cm inaccurate, because of the robot drifting past the point after releasing the motors. We ended up not taking the predefined area into account, it was used in the design phase to measure where the robot would be able to catch a thrown object, but it was not useable because its movement was too slow and the time used in data transfer compared to object flight time was a greater obstacle than expected. Thus it was not useful in the final product.

The robot should know where it is positioned

This requirement is fulfilled. The robot has variables of its current position as an X and Y coordinate set and its heading. These values are updated in the Arduino loop function using the encoders of the motors. When the robot reaches its destination the robot shuts down, but the servo motors drift, usually 11 cm, which is not considered to the current position of the robot. This would accumulate if the robot is not reset, and placed on the position $(0, 0)$ after every run.

The robot should be able to detect and track the thrown object

As the Kinect is used as a sensor for the robot, this requirement is fulfilled. The reason is because the Kinect is able to both detect and track the object. It is not able to detect every object thrown, and some times it detects other objects such as a human limb or something in the distortion in the room.
When the first impact point of the thrown object has been sent from the Kinect, it will keep tracking the object, but the robot will not receive further data, therefore the robot is not continuously tracking the object after the first impact point is received.

The robot should be able to calculate the impact point for the object

The Kinect sends the predicted impact point of the object to the robot. This impact point is not accurate, as mentioned before the robot is stopping 10-50 cm away from the predicted impact point. We would say that this requirement is partially fulfilled, because it can calculate an impact point, but not a precise impact point. This requirements success is scaled from how big of a bin is placed on top of the robot, and the size of the thrown object.

The robot should be able to move the trash bin, such that the thrown trash lands inside the bin

No trash bin has been attached to the robot, so this requirement is not fulfilled. If a trash bin had been attached to the robot, it should have a radius greater than 11 cm to catch the object, since 11 cm was the closest the robot was to the impact point.

The robot should be able to receive data from a computer, through a wireless network

This requirement is fulfilled. The computer connected to the Kinect is able to send the recorded data to the robot.

The system tasks should be able to be scheduled and verified

This requirement is fulfilled by the UPPAAL schedulability analysis as described in section 8.1 in the test chapter.

Conclusion

Considering the outcome of this project, the final product is a partial success. One can conclude that the hardware does not meet the time constraints of the problem, and the prediction is not reliable. When a user throws an object towards the robot, it is not able to catch the object, but it is able to give a prediction of the impact point, it is able to move the robot to the coordinates of the prediction, but will drift past the coordinates. The robot is able to position itself accordingly to the starting point, but it will not be able to continuously position itself after stopping at the impact point. The drift makes the robot unable to position correctly, so the robot has to be repositioned to the correct starting point after each use.

The robot is able to receive data from the Kinect sensor through a wireless network, and all the tasks for the system are schedulable, and the fact that this embedded system provides a basis for a solution to the problem concludes this as a successful system in our opinion. The biggest objective in this project was to learn about embedded systems and real time problems. Although this project did not concern many of the classical embedded systems problems, a cyclic executive was constructed and the analysis involved. Although considerations of memory management and cpu speed was not present, some considerations were performed. Even when considering the initial preconceived notions of the platform, motors and Kinect alot of unexpected problems arose, some of which were solved and others would need different hardware to solve the problem statement.

Future work 11

When the robot is at the impact point sent by the Kinect, it would release both of the motors and then drift further in the direction it was heading. The robot drifting should be avoided by making a brake function, making the robot stop at the exact point the Kinect sent.

An improvement to the current build would be to enable the robot to reset its position by returning to its starting position and resetting its heading, this could be done by implementing a function which achieves the described behaviour.

Further improvements would be to acquire new motors and wheels, but this could introduce scheduling problems since positioning is done through motor encoder interrupts, which could occur too often to schedule the tasks, if the motors are too fast. This is one of the problems of a cyclic executive, it is a very rigid construct which breaks if any of its scheduled tasks change, or new tasks are introduced.

The delimitations introduced In 5.2 could with further improvements be removed. The delimitation of multiple object tracking could be removed, by tracking all objects with the kinect sensor and sending information about all objects landing positions, but also the time in which they would get there. The robot then has to calculate the route it should take to catch all objects and which to skip if impossible to reach everything. The delimitation of not driving backwards could be solved by using motors that can turn around their own axis such that the robot always drives forward and doesn't need to turn around itself but rather just turn the wheels in the desired direction and go.

Another improvement would be testing the system for stack overflow, which wasn't tested, even though some of the theoretical aspects of it were discussed in 6.6.3, and also having the experience of using such tools.

If we were to introduce more real-time scheduling theories, the next step would be to implement graceful degradation for the wifi code. This would create the possibility of receiving data continuously, but the wifi code may not be scheduled to receive the data, if there is not enough time for the other tasks to execute. This is a possibility since it would only reduce accuracy of the prediction, since more data from the Kinect sensor leads to more precise prediction of the impact point. This would still introduce the problem of dual-shield compatibility, since the Wifi and Motor Shield cannot operate at the same time, due to them using the same pins.

The fact that both shields cannot operate at the same time could be eliminated by rerouting the pins on the Wifi Shield, or using a cable to send the data, instead of having a wireless connection. With a cable instead of the wireless connection would also reduce the time

used on sending the data, but also reduces the mobility of the robot.

Bibliography

Aamoth, **2013**. Doug Aamoth. *Finally, a Trash Can on Wheels That Automatically Catches Your Trash.* http://techland.time.com/2013/02/15/finally-a-trash-can-on-wheels-that-automatically-catches-your-trash/, 2013. Accessed: 16-12-2016.

Arduino.cc, **a**. Arduino.cc. *Arduino Motor Shield.* https://www.arduino.cc/en/Main/ArduinoMotorShieldR3. Accessed: 18-10-2016.

Arduino.cc, **b**. Arduino.cc. *Arduino WiFi Shield.* https://www.arduino.cc/en/Main/ArduinoWiFiShield. Accessed: 17-10-2016.

Arduino.cc, **c**. Arduino.cc. *WifiGuide.* https://www.arduino.cc/en/Guide/ArduinoWiFiShield. Accessed: 17-10-2016.

Bertolotti og Manduchi, **2012**. Ivan Cibrario Bertolotti og Gabriele Manduchi. *Real-Time Embedded Systems: Open-Source Operating systems Perspective.* ISBN: 9781439841549, ebook. CRC Press 2012, 2012.

Hurbain. Philippe "Philo" Hurbain. *LEGO® 9V Technic Motors compared characteristics.* http://www.philohome.com/motors/motorcomp.htm. Accessed: 12-10-2016.

MacEvoy et al. Warren MacEvoy, Matthew Murdoch, freenerd, John Macdonald, nicolaspanel og Matt Paine. *ArduinoUnit.* https://github.com/mmurdoch/arduinounit.

Regehr, **2006**. John Regehr. *Safe and Structured Use of Interrupts in Real-Time and Embedded Software.* https://www.cs.utah.edu/~regehr/papers/interrupt_chapter.pdf, 2006.

Rosenblattl og Wolf. Maximilian Rosenblattl og Andreas Wolf. *AVRFIX.* https://github.com/duckythescientist/avrfix. Accessed: 08-12-2016.

Sommerville, **2016**. Ian Sommerville. *Software Engineering tenth edition.* ISBN: 1-292-09613-6, Paperback. Pearson, 2016.

Techopedia.com. Techopedia.com. *Embedded Systems.* https://www.techopedia.com/definition/3636/embedded-system. Accessed: 25-09-2016.

Tidorum Ltd. Tidorum Ltd. *Bound-T.* http://www.bound-t.com/.

Uppsala University og Aalborg University. Uppsala University og Aalborg University. *UPPAAL.* http://www.uppaal.org/. Department of Information Technology at Uppsala University and Department of Computer Science at Aalborg University.

Wikipedia.com, **2016**. Wikipedia.com. *Kinect Wiki*.
`https://en.wikipedia.org/wiki/Kinect`, 2016. Accessed: 19-10-2016.

Increment one A

In this chapter the group will go through the requirements of the project and explain how these requirements can and has been implemented. The result of this would be a new set of requirements with some of the same requirements, new requirements and changed requirements.

A.1 Requirements

The requirements considered in this increment are marked with blue colouring.

- When a user throws an object towards the trash bin, within the predefined area, the bin should always catch the object
  - The robots predefined area should be calculated from the hardware limitations of the motors' speed
- The robot should know where it is positioned
  - The robot should have a starting position, from where it should be able to calculate it's current position
  - The robot's starting point should be placed outside its predefined area, such that it moves forward into the area
- The robot should be able to detect and track the thrown object
  - The thrown object should be detected and tracked by a Microsoft Kinect
  - The Kinect should send the coordinates of the impact point of the object to the robot
- The robot should be able to calculate the impact point for the object
  - Trajectory prediction should be used to calculate the impact point of the thrown object
- The robot should be able to move the trash bin, such that the thrown object lands inside the bin
  - The robot should be able to turn and drive forward and backwards
  - The robot should be able to recognize the coordinates sent from the Kinect
- The robot should be able to receive data from a computer, through a wireless network
- The system tasks should be able to be scheduled and verified

These requirements are rudimentary, and the very essence of the project lies within the fulfilment of these requirements. In this increment the fundamentals of the robots movement should be implemented, the predefined area should be defined, a starting point for this area should be determined and the Microsoft Kinect should be able to detect and track an object by using trajectory prediction.

A.2 System design

The following sections describe how the marked requirements in A.1 should be fulfilled. The theories and ideas will be explained subsequently.

A.2.1 Predefined area

The robots predefined area, is an area where the robot should catch the object within. This area is being made because of the limitations of the robots motors and is based on motor speed and average time of a throw, so the predefined area is where the robot should be expected to catch the object within.

A.2.2 Throwing

The reason why the throw of the object (which in this project will be a table tennis ball) is important, is because the robot should have as much time to move to the collision point as possible. Before the robot moves to the collision point, the Kinect should calculate the where the robot should move, send the data to the robot, and the robot then moves. The Kinect can at any point send new data to the robot, so its course have to be changed, therefore it is important for the robot to have sufficient time to move within the predefined area.

A.2.3 Microsoft Kinect

To detect the object, the Kinect has to use one of its cameras, more specifically the depth sensor. The depth sensor uses a range of infrared speckles that draw a pattern in the room. Every cluster of speckles can be identified from each other, which makes the Kinect able to differentiate between objects in the room. The depth of the specific object can be determined by the use of two cameras, as both cameras can identify the speckles at the object and then triangulate the distance between the cameras and the object. This speckle pattern technology will only work indoor, and limits the use to only one kinect, as the speckles can be washed out by other lightsources. [Wikipedia.com, 2016]

A.2.4 Trajectory prediction

When the object, referred to in this section as the projectile, is detected and the tracking of that projectile has started, the trajectory can be predicted. This prediction is limited to the amount of data sent by the sensory camera, meaning that for every camera reading, one detection of the object is gained. The precision of the prediction will increase according to the time a projectile has been tracked.
In this project the outdoor weather conditions, that might affect the projectile trajectory, are not considered, since the prediction is done indoor. As well, the effects of air resistance, also called drag, will not be considered.

The trajectory of a projectile is the path of a thrown projectile affected by gravity, without propulsion. To calculate a trajectory of a projectile, the initial height, the angle which the projectile is launced from, the speed of the projectile at launch and the gravitational acceleration must be taken into account.
The initial height in this project is the height at which the projectile is detected. The

angle and speed at which the projectile is launched, will be calculated from the first few trackings after detecting the projectile. The gravitational acceleration is considered as $9.81m/s^2$, which is the common value near the earths surface.

To catch the projectile, the distance and time of flight have to be calculated. This is done with two mathematical formulas:
g : *the gravitational acceleration* ($9.81m/s^2$)
θ : *the angle at launch*v : *the speed at launch*
y_0 : *the initial height*
d : *the total horizontal distance traveled*
t : *the time of flight*
v_{vert} : *the vertical velocity*
v_{hori} : *the horizontal velocity*
t_h : *the time since first detection*
d_t : *the distance at time t*

The above variables can express the formulas needed to provide the total distance traveled by the projectile, and the time of flight.

Distance traveled:

$$d = \frac{v\ cos\ \theta}{g}(v\ sin\ \theta + \sqrt{(v\ sin\ \theta)^2\ +\ 2gy_0})$$

Time of flight:

$$t = \frac{d}{v\ cos\ \theta} = \frac{v\ sin\ \theta\ +\ \sqrt{(v\ sin\ \theta)^2\ +\ 2gy_0}}{g}$$

To predict the trajectory of the projectile, the altitude and distance of the projectile at any time during the flight must be calculated, according to the initial height (y_0):

$$y = v_{vert}t - \frac{1}{2}gt^2$$

$$d_t = v_{hori}t$$

After this section, the projectile can be tracked, and the path for a given projectile can be calculated to a certain degree of precision.

A.2.5 Gyroscope and Accelerometer

For positioning the robot, the Lego NXT Gyro and Accelerometer sensors will be considered. These sensors will be used together to position the robot, as the gyroscope will tell the number of degrees the robot has turned, which is relative to the heading of the robot at the first recording of data. This heading will be used together with an

accelerometer, which can be used to calculate the distance the robot has traveled since the first recording of data. The speed, travel time and heading will be the data of these sensors.

A.2.6 Movement

The robot is expected to be able to drive both forward and backwards. The robot is expected to turn using either one active wheel, or using both wheels in a counterclockwise motion in order to turn faster.

A.3 Implementation

This section explains how the marked requirements has been implemented and whether or not the group changed the requirement, as something didn't work out as expected.

A.3.1 Gyroscope and Accelerometer in use

For the implementation of the gyroscope and the accelerometer, a test has to be done, to benchmark the precision of the sensors. For this, the data sent from the Arduino when the sensor has been plugged into the Arduino, was plotted in the Serial Plotter, which is a feature in the Arduino IDE.

The accelerometer would not be precise enough to calculate the distance the robot had moved. The accelerometer would give an acceleration, to calculate from an acceleration to a distance, the time had to be multiplied to the acceleration twice. If the acceleration was not precise, this would lead to a big margin of error.

The gyroscope worked just fine, but when it was not moving, the data seen in the Serial plotter would wander. This meant that when reading data from the gyroscope, either the gyroscope should be reset before reading data or the value would be the difference of when it started reading till it stopped.

Instead of using the gyroscope and the accelerometer, it was decided to use the motor' encoderes, which can be used to calculate the heading and the distance moved.

A.3.2 Predefined area

The robots predefined area will be calculated using the time of flight and the speed of the motors. This predefined area is limited to be strictly in front of the robot. A bouncing throw would have a travel time of 2 - 2.15 seconds, explained in greater detail in section A.3.3. The predefined area was constructed using the robot itself. The robot was placed marking its starting point, and was set to turn a specific number of degrees and move forward for two seconds. After a series of runs, the area marked by the robot turned out as shown in figure A.1. The predefined area measures 2851.5cm^2.

A.3.3 Throwing

To ensure that the robot is provided the maximum amount of time for calculations and movement, the average time of flight of an object at a designated distance was measured.

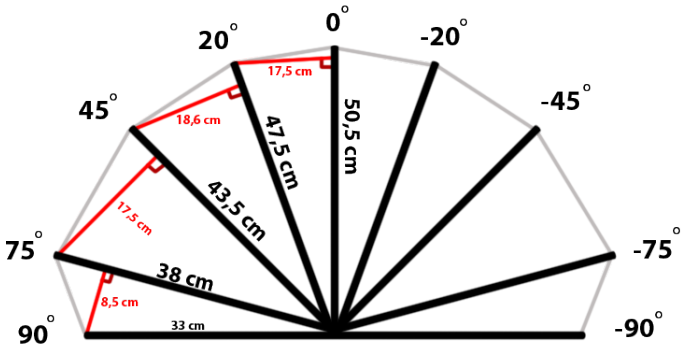

Figure A.1. The robots predefined area

At a distance of three meters, the average time of flight of the object was 1.1 seconds, and at a distance of five meters, the average time of flight was 1.25 seconds.

With the results from the abovementioned measurements, it is concluded that a regular throw would not provide enough time for the robot to catch the object. As mentioned in the 4.4.2 section, the robots speed was calculated to be 87 RPM. With a 1.1 seconds time of flight, with a distance of three meters, the impact point should be within a 280 mm radius from the robot. This simply is not fast enough to catch the object from a three or five meter distance.

This introduces the idea of bouncing the object, in order to solve the above mentioned problem. The idea is, that the initial height of the object is considered to be the floor instead of the top half of the arc, which in theory should provide more time for the robot. After conducting another set of tests, this theory proved to be correct. The average time of flight at a distance of three meters is 2 seconds, and at a distance of five meters the average time of flight is on average 2.15 seconds. The bouncing throw provides the robot with close to double the amount of time.

A.3.4 Movement

An Arduino Motor Shield is used to enable greater control of the movement of the motors through the motor shields library called Adafruit which is included through #include <AFMotor.h> (unsure how to present this nicely) The Arduino Motor Shield and library enables manipulation of the individual wheels instead of only being able to either rotate forward or backwards on both wheels simultaneously. The wheels were tested and forward motion was achieved with a speed of roughly 25.5 centimetres per second. It was discovered that the speed at which the robot drives backwards is much slower than its forward moving speed, because of the speed of backwards motion, it was decided to not consider driving backwards at all since the speed were inadequate and since this would also simplify the problem, and could later be solved with different motors. To solve this issue, it was decided

to change the throw to always be in front of the machine, thus ensuring it would never be forced to drive backwards. This does however not take into account if the Kinect projectile prediction predicts the projectile will land behind the machine but this issue only arises if prediction is sufficiently miscalculated which is unlikely.

A.4 Evaluation

This section will include a short summary of the chapter, ending up with a new requirement list, based on the how the group implemented the marked requirements from the beginning of the chapter.

The requirements stating that the robot should know where it is positioned, will persist through this increment, as no solution was found in this increment. The use of a gyroscope and accelerometer was not the right solution for this project, as the need of precise data would not be supported by sensors with a large amount of jitter. Another way of positioning the robot would be by calculating the distance traveled on each wheel, through the motor encoder in the NXT servo motors. The distance traveled would be calculated by the rotation of the wheel and the circumference of that wheel. The robots starting position will always determine where the predefined area of the robot is along with its hardware limitations. The concept of a predefined area is entirely based on the rotation speed of each motor, the wheel circumference and the duration of the throw. If any element that defines the predefined area changes, then so does the predefined area. Movement was not nearly as simple as expected and the implications of the problems encountered in its implementation is not fully solved in this increment.

The requirement of driving forward and backwards is redefined as only consisting of driving fowards. The reason for the change was explained in the section A.2.6.

The requirements after finalizing increment one is:

- The trash bin should catch the trash if the user throws it towards the trash bin and within a predefined area
  - The robots predefined area should be calculated from the hardware limitations of the motors' speed
- The robot should know where it is positioned
  - The robot should have a starting position, from where it should be able to calculate it's current position through calculations of the motor encoders
  - The robot's starting point should be placed outside its predefined area, such that it moves forward into the area
- The robot should be able to detect and track the thrown trash
  - The thrown trash should be detected and tracked by a Microsoft Kinect
  - The Kinect should send the coordinates of the impact point of the trash to the robot
- The robot should be able to calculate the impact point for the object
  - Trajectory prediction should be used to calculate impact point of the thrown trash

- The robot should be able to move the trash bin, such that the thrown trash lands inside the bin
  - The robot should be able to turn and drive forward
  - The robot should be able to recognize the coordinates sent from the Kinect
- The robot should be able to receive data from a computer, through a wireless network
- The system tasks should be able to be scheduled and verified

Increment two B

This chapter is made with the same procedure as Increment one. The results from increment one will be discussed and implemented, this should result in the same, new or changed requirements.

B.1 Requirements

The requirement marked with blue will be considered throughout this chapter.

- The trash bin should catch the trash if the user throws it towards the trash bin and within a predefined area
  - The robots predefined area should be calculated from the hardware limitations of the motors' speed
- The robot should know where it is positioned
  - The robot should have a starting position, from where it should be able to calculate it's current position through calculations of the motor encoders
  - The robot's starting point should be placed outside its predefined area, such that it moves forward into the area
- The robot should be able to detect and track the thrown trash
  - The thrown trash should be detected and tracked by a Microsoft Kinect
  - The Kinect should send the coordinates of the impact point of the trash to the robot
- The robot should be able to calculate the impact point for the object
  - Trajectory prediction should be used to calculate impact point of the thrown trash
- The robot should be able to move the trash bin, such that the thrown trash lands inside the bin
  - The robot should be able to turn and drive forward
  - The robot should be able to recognize the coordinates sent from the Kinect
- The robot should be able to receive data from a computer, through a wireless network
- The system tasks should be able to be scheduled and verified

B.2 System design

The following section will describe and explain how the marked requirements from section B.1 will be fulfilled.

B.2.1 Wifi Shield

When the Kinect have found the coordinates for the collision point of the thrown object, these coordinates should be sent to the robot, for it to be able to move to the collision point and catch the object. The data should be sent, as earlier mentioned in section 4.4.3, to the Arduino wifi shield. The Arduino and the computer running the Kinect program should be connected to their own network, with the computer sending data to the wifi shields IP-address.

B.2.2 Robot positioning

In the previous chapter, it was decided to use the motor's encoders to calculate the heading of the robot and the distance it has moved. The encoders will keep track of the number of degrees the specific motor have turned, which can be calculated to a distance using the wheels circumference. The coordinate set for the collision point of the object and the robots current position, will be used to calculate the heading, which the robot should have to reach the collision point.

B.3 Implementation

This section explains how the marked requirements were actually implemented in the project and whether or not the requirements have been changed, removed or been fulfilled.

B.3.1 Wifi Shield

To implement the Wifi Shield as part of the Arduino, the router connecting the Wifi Shield with a computer had to be port forwarded. The private network hosted by the D-link router had a specific ssid and password, which had to be explicitly assigned in the Arduino code for the Wifi Shield. The setup in the Arduino code was used to start the connection between the computer and the Wifi Shield, as well as printing the log of the connection. The setup for the wifi shield can be found in listing B.1.

Listing B.1. Connecting the Wifi shield to the network

```
1  void setup() {
2         Serial.begin(9600);
3
4         Serial.println("Attempting to connect to WPA network...");
5         status = WiFi.begin(ssid, pass);
6
7         if ( status != WL_CONNECTED) {
8                 Serial.println("Couldn't get a wifi connection");
9                 while(true);
10        }
11        else {
12                server.begin();
13                Serial.println("Connected to network");
14        }
15        ip = WiFi.localIP();
16        Serial.println(ip);
17 }
```

After connecting the Wifi Shield to the router, the Arduino is ready to receive data. The data will be sent from a laptop, which connects and sends data as shown in B.2. A

tcpClient is created and connected to the IP of the Wifi Shield, and the port that has been forwarded. The data sent in the code is a string "test", which is sent as a byte-array, and after sending the data, the tcpClient closes the connection.

Listing B.2. Connecting the computer to the Wifi Shield

```
1 tcpClient.Connect("192.168.0.100", 9999);
2 string data = "test";
3 const int IntSize = 4;
4 byte[] bytedata;
5 Stream stream = tcpClient.GetStream();
6
7 stream.Write(Encoding.UTF8.GetBytes(data.ToCharArray()), 0, data.Length);
8 tcpClient.Close();
```

When both the computer and the Wifi shield has made the connection, and the computer has sent the data, the Arduino can then receive the data. This is done through a Wifi client, from the Arduino Wifi library. When assigning the client to the server.available, the value of the client in an if-statement would be true if any client from the server has data available for reading, or false if no data was available. If a client is connected, and the client has data available for reading, the next byte will be appended on the string readString, and after all the data available has been read, the full string sent by the client will be printed.

Listing B.3. Receiving data from the computer

```
1  client = server.available();
2  while (client.connected()){
3         if (client.available()) {
4                 char c = client.read();
5                 readString += c;
6         }
7  }
8  if(readString != ""){
9         Serial.println(readString);
10        readString = "";
11 }
```

B.3.2 Robot positioning

In the top of the Arduino program, a lot of variables were declared, which can be seen in listing B.4. The two volatile integers leftTotal and rightTotal are used to count the number of degrees the motor have turned. leftTotal and rightTotal are volatile integers because the value of the variables can be changed by other things than the code, which in this case is the motors.
motorLeftRun and motorRighRun are booleans used to check whether the motors are running or not.
The variable called heading have the value of the robot's heading, which in this project will be the degrees the robot's direction is pointing away from the y-axis of its predefined area. Then robot's current position will be saved as coordinates in the variables posX and posY.
Several integers are declared which will get further explanation later in the section. Also in the top of the program some identifiers have been defined and the necessary libraries have been included to the program.

Listing B.4. The defined and declared variables

```
1  volatile int leftTotal = 0;
2  volatile int rightTotal = 0;
3  bool motorLeftRun = false;
4  bool motorRightRun = false;
5  double heading = 0.0, posX = 0.0, posY = 0.0;
6  unsigned int tid;
7  int leftTemp, rightTemp, goalX = 230, goalY = 330, margin = 10, loopcount = 0, signalCount = 0;
```

An arduino program is split up into a setup function and a loop function, where the code in the loop function should describe the robot's behaviour. The setup function are run once for every power up or reset of the Arduino board. The setup function can be found in listing B.5. Two pins are setup as inputs for the arduino board and an interrupt is attached to each of them. Every time a change occurs for the input pins, which in this case is every time the motors have turned 1 degree, the functions incrementLeft and incrementRight are called accordingly. The two functions increments the counters leftTotal and rightTotal which are counting the number of degrees the motors have turned.
Next up the motors are turned on. This is done by setting their speed, which here is set to the highest speed possible, and then both motors are ret to RELEASE, which will stop the motors. Also the booleans motorLeftRun and motorRightRun are set to false.
At the bottom of the setup funtion, then leftTotal's and the rightTotal's values have been assigned to the temporary variables leftTemp and rightTemp.

Listing B.5. The setup function

```
1  pinMode(LEFTENCODERPIN, INPUT);
2  attachInterrupt(LEFTPINTOINTERRUPT, incrementLeft, CHANGE);
3  pinMode(RIGHTENCODERPIN, INPUT);
4  attachInterrupt(RIGHTPINTOINTERRUPT, incrementRight, CHANGE);
5
6  motorLeft.setSpeed(255);
7  motorRight.setSpeed(255);
8
9  motorLeft.run(RELEASE);
10 motorLeftRun = false;
11 motorRight.run(RELEASE);
12 motorRightRun = false;
13
14 leftTemp = leftTotal;
15 rightTemp = rightTotal;
```

When the arduino program enters the loop function, seen in listing B.6, the first thing it will do is to check the if-statement at line. This if-statement checks if the robot is at its desired position, which is the coordinate set sent from the Kinect(goalX and goalY). If the robot was at the position it will release both motors and exit the program, else it will skip this if statement and continue in the loop function.
In lines 9-12 the difference of the desired coordinate and the current coordinate of the robot are calculated and assigned to the variables deltaY and deltaX. With these new variables the heading of which the robot should have to hit the desired point. The delta heading are calculated using the goalHeading and the current heading of the robot.
In lines 14 - 23 the if-else statement checks if the robot is on the right heading. If the robot's heading is less than -0.1 it will release the right motor and drive forward with the left, making the robot turn right. The robot will do the same check for 0.1 and will release the left motor and drive forward on the right, making the robot turn left. If the robot is

on its right heading, it will drive forward, towards it desired point.
In lines 25-35 two new integers, currentLeft and currentRight, are assigned the values of leftTotal and rightTotal. Then the distance of how much the wheels have moved since last time it was looped through is calculated. This is done by taking the difference of currentLeft and leftTemp times DISTPRDEGREE, which was defined at the beginning of the program, the same is done with currentRight and tempRight. Then leftTemp and rightTemp are assigned the values of currentLeft and currentRight.
The distance the robot have moved is the average of how much the two motors have moved, so the the distances of the two motors are added together and divided by two. With the distance, the robots new current position is calculated assigned to the values posX and posY, and the new heading for the robot.

Listing B.6. The loop function

```
1  void loop() {
2         if(round(posX) <= goalX + margin && round(posX) >= goalX - margin && round(posY) <= goalY +
               margin && round(posY) >= goalY - margin){
3                 motorLeft.run(RELEASE);
4                 motorRight.run(RELEASE);
5                 delay(50);
6                 exit(0);
7         }
8
9         double deltaX = goalX - posX;
10        double deltaY = goalY - posY;
11        double goalHeading = atan(deltaX/deltaY);
12        double deltaHeading = goalHeading - heading;
13
14        if(deltaHeading < -0.1){
15                motorLeft.run(FORWARD);
16                motorRight.run(RELEASE);
17        }else if(deltaHeading > 0.1){
18                motorLeft.run(RELEASE);
19                motorRight.run(FORWARD);
20        }else{
21                motorLeft.run(FORWARD);
22                motorRight.run(FORWARD);
23        }
24
25        delay(10);
26        int currentLeft = leftTotal;
27        int currentRight = rightTotal;
28        double deltaLeft = (currentLeft - leftTemp) * DISTPRDEGREE;
29        double deltaRight = (currentRight - rightTemp) * DISTPRDEGREE;
30        leftTemp = currentLeft;
31        rightTemp = currentRight;
32        double dist = (deltaLeft + deltaRight) / 2.0;
33        posX += (dist * sin(heading));
34        posY += (dist * cos(heading));
35        heading += (atan((deltaRight - deltaLeft) / WHEELDIST));
36 }
```

B.4 Evaluation

This section will include a short summary of the chapter and conclude with an updated requirement list, showing changes from the first requirement list in the chapter.

This increment focused on two things: The robot's positioning and movement and connecting the robot with the Kinect program.

Using the Motor encoders it is now possible to calculate the heading and the distance moved, making it possible for the robot to drive to a known location, which is the coordinate point sent by the Kinect.
The connecting between the robot and the Kinect program have been established on their own private network. The robot will connect to the network and will wait for the Kinect program to sent data to its IP address.

This is the requirements after finalizing increment two:

- The trash bin should catch the trash if the user throws it towards the trash bin and within a predefined area
  - The robots predefined area should be calculated from the hardware limitations of the motors' speed
- The robot should know where it is positioned
  - The robot should have a starting position, from where it should be able to calculate it's current position through calculations of the motor encoders
  - The robot's starting point should be placed outside its predefined area, such that it moves forward into the area
- The robot should be able to detect and track the thrown trash
  - The thrown trash should be detected and tracked by a Microsoft Kinect
  - The Kinect should send the coordinates of the impact point of the trash to the robot
- The robot should be able to calculate the impact point for the object
  - Trajectory prediction should be used to calculate impact point of the thrown trash
- The robot should be able to move the trash bin, such that the thrown trash lands inside the bin
  - The robot should be able to turn and drive forward
  - The robot should be able to recognize the coordinates sent from the Kinect
- The robot should be able to receive data from a computer, through a wireless network
- The system tasks should be able to be scheduled and verified

Increment three C

The procedure of this increment will be the same as the previous ones. The requirements from section B.4 in increment two, will be the base for this increment. The requirements in focus will be discussed, tried implemented and the evaluated upon, which might lead to changes of the requirements list.

C.1 Requirements

- The trash bin should catch the trash if the user throws it towards the trash bin and within a predefined area
  - The robots predefined area should be calculated from the hardware limitations of the motors' speed
- The robot should know where it is positioned
  - The robot should have a starting position, from where it should be able to calculate it's current position through calculations of the motor encoders
  - The robot's starting point should be placed outside its predefined area, such that it moves forward into the area
- The robot should be able to detect and track the thrown trash
  - The thrown trash should be detected and tracked by a Microsoft Kinect
  - The Kinect should send the coordinates of the impact point of the trash to the robot
- The robot should be able to calculate the impact point for the object
  - Trajectory prediction should be used to calculate impact point of the thrown trash
- The robot should be able to move the trash bin, such that the thrown trash lands inside the bin
  - The robot should be able to turn and drive forward
  - The robot should be able to recognize the coordinates sent from the Kinect
- The robot should be able to receive data from a computer, through a wireless network
- The system tasks should be able to be scheduled and verified

C.2 System design

This section will describe and explain the marked requirements, for how they are planned to be fulfilled.

C.2.1 Connecting Arduino and Kinect

For connecting the Arduino and Kinect, the Wifi library is used as explained in Appendix 6.5. The Kinect should create a TCP-client, which should send the appropriate data to the Arduino. This TCP-client should connect when it can send data, and disconnect after sending the data, since the Arduino has a timer that closes the connection after 10 seconds of inactivity. The sent data should be of the form (x, z) expressing the distance from the kinect to the impact point of the object and the ground. The x-coordinate should express the horizontal position, and the z-coodinate should express the depth position. As the Kinect camera should be position and tilted so that the bottom angle of the cameras point of view is parallel to the ground, makes the y coordinate useless, as the impact point would always be at (x, (y = 0), z). This delimitation also gives the camera a sense of where the ground is, since it would not be able to see the ground, and therefore would calculate a different impact point, as the object would in a sense fall through the ground in the picture.

C.2.2 Scheduling

The calculation for the WCET of the system is described in chapter 7 section 7.3.

The WCET for the two functions and the time spent on sending data from the Kinect via wifi ,using the Micros() function, is listed below:

- updatePosAndHead = 1076 microseconds
- driveTowardsGoal = 732 microseconds
- WiFi = 8433 microseconds

The above results will be used for a schedulability analysis made in UPPAAL in section C.3.2.

C.3 Implementation

This section will describe if the requirements were fulfilled through implementation.

C.3.1 Connecting Arduino and Kinect

When the Kinect sensor has spotted the object in several frames, it returns a function that gives the objects position in relation to a given time in the future. The position given is in three dimensions, but only the x and y-coordinate are of interest, because the coordinate is only returned when we know the object must have hit the ground. The x-coordinate is given in meters, in relation to the Kinects left field of view. The y-coordinate is also given in meters, and is the distance between the balls impact spot with the ground, and the Kinect itself. This can be seen in C.1

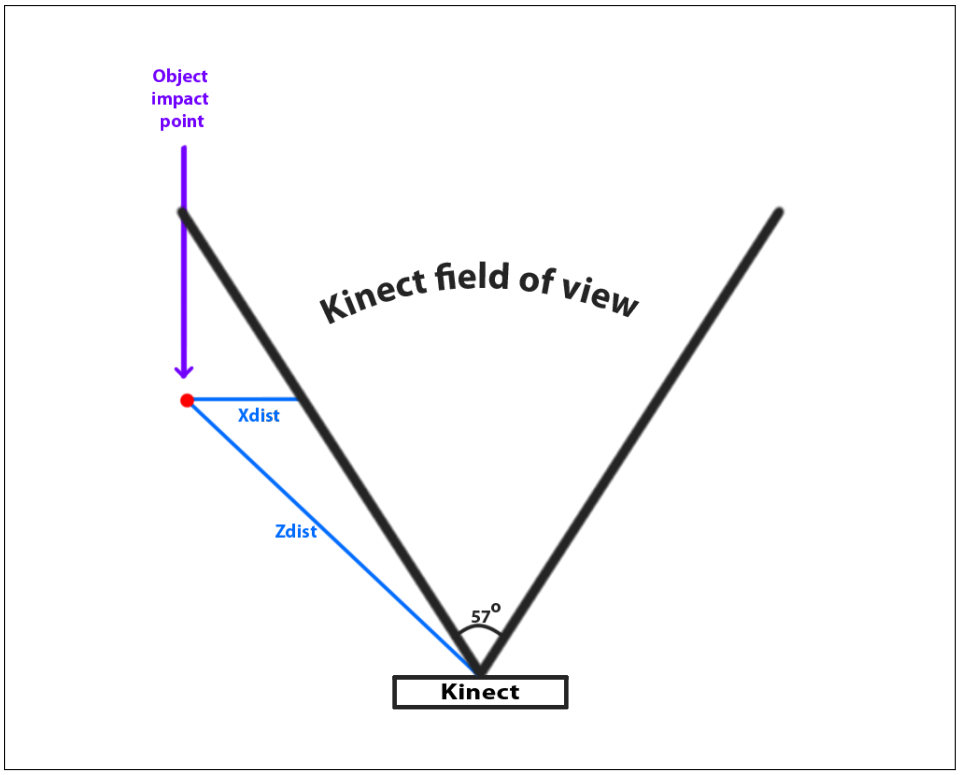

Figure C.1. Object impact in relation to Kinect

This information is not useful to the Arduino as is, so the coordinate must be converted to a coordinate that is more useful to the Arduino. Looking at C.1, the length xdist represents the distance from the Kinects field of view to the impact point of the object, and the zdist represents the distance between the Kinect and the impact point of the object. This is the information given by the Kinect library. Since this coordinate system is not straight in relation to the Kinect and changes from impact to impact, depending on the depth, we first calculate the impact point as shown by a and b on C.2.

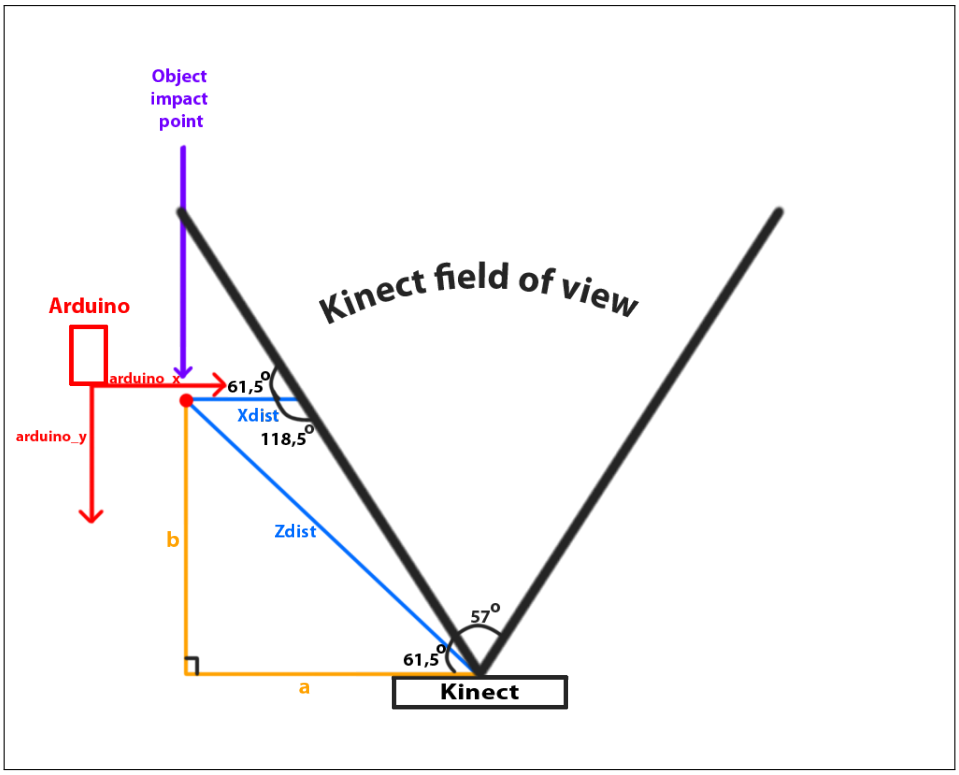

Figure C.2. Object impact in relation to Kinect with calculated a and b, and the Arduino

The code for calculating the coordinates using trigonometry, where XSquare is a and YSquare is b on figure:

Listing C.1. Converting coordinates form Kinect

```
1  double as1 = (Math.Sin(2.06821516));
2  double as2 = as1 * Math.Abs(Xdist);
3  double as3 = (ZDist);
4  double as4 = as2 / as3;
5  double A = Math.Asin(as4);
6  A = A * 180 / Math.PI;
7  double B = 180 - A - 118.5;
8  double b = ((Math.Sin(B * Math.PI / 180)) * Math.Abs(v3.X)) / (Math.Sin(A * Math.PI / 180));
9  double A2 = 90 - B;
10 double XSquare = ((Math.Sin(A2 * Math.PI / 180)) * (v3.Z - 0.5)) / (Math.Sin(90 * Math.PI / 180));
11 double YSquare = ((Math.Sin(B * Math.PI / 180)) * (v3.Z - 0.5)) / (Math.Sin(90 * Math.PI / 180));
```

Using the calculated a and b, it is only a matter of addition and subtraction, to match the Arduinos coordinate system, based on its distance from the Kinect.

The coordinate is the send to the Arduino, using a TCP connection. The code for this is shown here:

Listing C.2. Sending data to the Arduino

```
1  tcpClient = new TcpClient();
2  tcpClient.Connect("192.168.0.100", 9999);
3  stream = tcpClient.GetStream();
4
5  stream.Write(Encoding.UTF8.GetBytes((finalx.ToString() + ";" + finaly.ToString()).ToCharArray()),
       0, (finalx.ToString() + ";" + finaly.ToString()).Length);
6  tcpClient.Close();
```

A new TCP connection is opened by using the Arduinos IP-address, which is then used, by writing to its stream. When this code has executed, the Arduino has the information ready.

C.3.2 UPPAAL model of schedulability

To create the UPPAAL model and to check the schedulability of the tasks used for Dumpsty, every tasks WCET is calculated. This execution time is attained through running the code multiple times with a set timer, and writing the timer in a log after executing the specific task. There are four tasks for Dumpsty: Updating the position and heading (referred to as "PrA"), driving towards the goal position ("PrB"), the WiFi code that has to be looped ("PrC") and the delay to not execute the tasks seven times within the deadline ("PrD"). Here are the WCET attained through the tests, and to the right of the clock cycles is the WCET used in UPPAAL. The WCET used in UPPAAL has a significant margin greater than the WCET gained through the tests, since there is enough time to give each task a greater WCET, and the worst case tested is not necessarily the WCET of the tasks.

- PrA Tested: 1067 microseconds UPPAAL: 2 milliseconds
- PrB Tested: 732 microseconds UPPAAL: 1 milliseconds
- PrC Tested: 8469 microseconds UPPAAL: 9 milliseconds
- PrD Tested: 10000 microseconds UPPAAL: 10 milliseconds

The interrupt generated by the motor encoders is taken into account in every task, but to simplify the schedulability analysis, the WCET of the interrupt has been multiplied to the WCET of every greater task. Therefore the interrupt and the interrupt handler is not considered as a task, but as a part of the other tasks.

When describing these tasks in UPPAAL, two different classes were declared: One to instantiate a cyclic executive task instance of every task, and a CPU class to control the synchronization between all the tasks, as shown in the following figure:

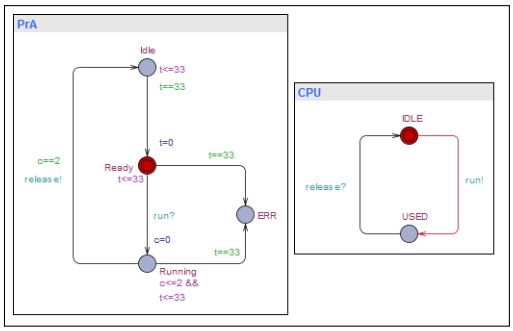

Figure C.3. Automata in UPPAAL

When the automata of the tasks has been created, every task has to be declared with an ID, a deadline and a period. The deadline has been declared for every task to be within the MIT of coordinates from the Kinect sensor, which is 33 milliseconds. The tasks are declared as:

- PrA = TASK(1, 33, 2);

- PrB = TASK(2, 33, 1);
- PrC = TASK(3, 33, 9);
- PrD = TASK(4, 33, 10);

After creating the automata and declaring the tasks, the UPPAAL schedulability analysis can be done. This is done through the verifier in UPPAAL, with two different queries:

Listing C.3. Queries for UPPAAL

```
1 E<> PrA.Ready and PrB.Ready and PrC.Ready and PrA.t==0 and PrB.t==0 and PrC.t==0 and time>0
2 E[] not (PrA.ERR or PrB.ERR or PrC.ERR)
```

The first query dictates that each tasks instance should be in a ready state, each tasks should have completed within the deadline and some interval of time has to be passed. This creates a scenario where the tasks has all been executed, and none of them have exceeded the deadline. The second query dictates that no task should ever enter an error state. After verifying both queries it is possible to say whether the tasks can be scheduled or not, in this case it is possible.

C.4 Evaluation

Many different methods were used to try to calculate the WCET for the systems functions. At first it was tried to count the clock cycles from the complied assembly code, which proved to be very difficult, since the bounds of the libraries was unknown. Because of these problems third party software and libraries was used to calculate the WCET, but the once again without any success. The method used to calculate the WCET was to use the micros() function, available in the arduino IDE, to measure the time used on the function, which would give close to a WCET of the functions. The tasks all had a significant margin to improve the possibility of hitting actual WCET.
The WCET's were used with the software UPPALL, as described in section C.3.2. This concluded that the tasks was easily schedulable within the defined deadlines.

After finalizing increment three the requirements in the following list will be the final requirements for the system:

- The trash bin should catch the trash if the user throws it towards the trash bin and within a predefined area
  - The robots predefined area should be calculated from the hardware limitations of the motors' speed
- The robot should know where it is positioned
  - The robot should have a starting position, from where it should be able to calculate it's current position through calculations of the motor encoders
  - The robot's starting point should be placed outside its predefined area, such that it moves forward into the area
- The robot should be able to detect and track the thrown trash
  - The thrown trash should be detected and tracked by a Microsoft Kinect
  - The Kinect should send the coordinates of the impact point of the trash to the robot
- The robot should be able to calculate the impact point for the object

  - Trajectory prediction should be used to calculate impact point of the thrown trash
- The robot should be able to move the trash bin, such that the thrown trash lands inside the bin
  - The robot should be able to turn and drive forward
  - The robot should be able to recognize the coordinates sent from the Kinect
- The robot should be able to receive data from a computer, through a wireless network
- The system tasks should be able to be scheduled and verified

Clock cycles D

This appendix includes all the instructions from the Math.h and AFMotor.h libraries used to make the functions for the arduino program. These have all been counted by looking at the assembly code of the program. Many of the instructions have loops which is denoted with parentheses and a star, such as (7)* will be a loop of 7 clock cycles.

- __fp_Split3 = 41
- __fp_round = 26
- __fp_pscA = 11
- __fp_pscB = 11
- __fp_nan = 7
- __fp_inf = 10
- __fp_Zero = 11
- __fp_szero = 10
- __addsf3x = 113 + (7)*
- __addsf3 = 147 + (7)*
- __fp_cmp = 41
- __cmpsf2 = 53
- __subsf3 = 148 + (7)*
- __gesf2 = 53
- digitalWrite = 82
- M.latch_tx = 572 + (290 + (5)*)*
- M.run = 634 + (5)* + (290 + (5)*)*
- delay = 52 + (6) + (51 + (6)*)*
- __divsf3x = 229 + (17)* + (13)* + (33 + (17)*)* + (8)*
- __divsf3 = 261 + (17)* + (13)* + (33 + (17)*)* + (8)*
- inverse = 269 + (17)* + (13)* + (33 + (17)*)* + (8)*
- interrupt = 40
- __mulsf3x = 129 + (13)*
- __mulsf3 = 161 + (13)*
- square = 163 + (13)*
- Powser = 130 + (174 + (13)*)* + (231 + (13)*)*
- atan = 989 + (13)* + (174 + (13)*)* + (231 + (13)*)* + (17)* + (8)* + (33 + (17)*)* + (7)*
- floatisf = 46 + (11)*
- __fp_splitA = 17
- __fp_rempio2 = 91 + (24)* + (10)*
- __fp_mpack = 22
- __fp_powsodd = 461 + (13)* + (174 + (13)*)* + (231 + (13)*)*

- __fp_sinus = 629 + (13)* + (174 + (13)*)* + (231 + (13)*)* + (7)*
- sin = 734 + (13)* + (174 + (13)*)* + (231 + (13)*)* + (7)* + (10)* + (24)*
- cos = 727 + (13)* + (174 + (13)*)* + (231 + (13)*)* + (17)* + (10)* + (24)*
- driveTowardsGoal = 4185 + (13)* + (174 + (13)*)* + (231 + (13)*)* + (17)* + (8)* + (33 + (17)*)* + (7)* + (5)* + (290 + (5)*)*
- updatePosAndHead = 3560 + (11)* + (13)* + (7)* + (174 + (13)*)* + (231 + (13)*)* (10)* + (24)* + (17)* + (33 + (17)*)*

Arduino code E

The following listing is the arduino code for this project.

Listing E.1. Finished arduino code for the project

```
#define LEFTENCODERPIN 20
#define RIGHTENCODERPIN 21
#define WHEELDIST 124
#define DISTPRDEGREE 0.48869219055

#include <AFMotor.h>
#include <Math.h>
#include <WiFi.h>

char ssid[] = "Dumpsty";  //  your network SSID (name)
char pass[] = "test1234"; // your network password
int status = WL_IDLE_STATUS;  // the Wifi radio's status
IPAddress ip;
WiFiServer server(9999);
WiFiClient client;
String dataString = "";


volatile int leftTotal = 0;
volatile int rightTotal = 0;
double heading = 0.0, posX = 0.0, posY = 0.0, margin = 10.0, goalX = 0.0, goalY = 0.0;
int leftTemp, rightTemp;

void setup() {
	Serial.begin(9600);
	status = WiFi.begin(ssid, pass);
	if ( status != WL_CONNECTED) {
		exit(0);
	}
	else {
		server.begin();
	}
	ip = WiFi.localIP();

	while(dataString == ""){
		client = server.available();
		if (client.connected()){
			while (client.available()) {
				char c = client.read();
				dataString += c;
			}
		}
	}
	goalX = dataString.substring(0,dataString.indexOf(';')).toInt();
	goalY = dataString.substring(dataString.indexOf(';')+1).toInt();

	pinMode(LEFTENCODERPIN, INPUT);
	attachInterrupt(3, incrementLeft, CHANGE);
	pinMode(RIGHTENCODERPIN, INPUT);
	attachInterrupt(2, incrementRight, CHANGE);
```

```
51
52          leftTemp = leftTotal;
53          rightTemp = rightTotal;
54  }
55
56  void loop() {
57          t = millis();
58          wifi();
59          updatePosAndHead();
60          driveTowardsGoal();
61          delay(10);
62          updatePosAndHead();
63          driveTowardsGoal();
64          while(micros()-t<33){}
65  }
66
67  void wifi(){
68          delay(9);
69  }
70
71  void updatePosAndHead(){
72          int currentLeft = leftTotal;
73          int currentRight = rightTotal;
74          double deltaLeft = (currentLeft - leftTemp) * DISTPRDEGREE;
75          double deltaRight = (currentRight - rightTemp) * DISTPRDEGREE;
76          leftTemp = currentLeft;
77          rightTemp = currentRight;
78          double dist = (deltaLeft + deltaRight) / 2.0;
79          posX += (dist * sin(heading));
80          posY += (dist * cos(heading));
81          heading += (atan((deltaRight - deltaLeft) / WHEELDIST));
82  }
83
84  void driveTowardsGoal(){
85          AF_DCMotor motorLeft(1);
86          AF_DCMotor motorRight(2);
87
88          motorLeft.setSpeed(255);
89          motorRight.setSpeed(255);
90
91          if(posX <= goalX + margin && posX >= goalX - margin && posY <= goalY + margin && posY >=
                goalY - margin){
92                  motorLeft.run(RELEASE);
93                  motorRight.run(RELEASE);
94                  delay(50);
95                  exit(0);
96          } else {
97                  double deltaX = goalX - posX;
98                  double deltaY = goalY - posY;
99                  double actualGoalHeading;
100                 if(deltaX == 0 && deltaY > 0)
101                         actualGoalHeading = 0;
102                 else if(deltaX == 0 && deltaY < 0)
103                         actualGoalHeading = PI;
104                 else
105                         actualGoalHeading = deltaX >= 0 ? atan(deltaY/deltaX) - (PI/2) :
                                atan(deltaY/deltaX) + (PI/2);
106
107                 double deltaHeading = actualGoalHeading - heading;
108                 if(deltaHeading > PI)
109                         deltaHeading -= 2*PI;
110                 else if(deltaHeading < -PI)
111                         deltaHeading += 2*PI;
112                 if(deltaHeading < -0.1){
113                         motorLeft.run(FORWARD);
114                         motorRight.run(RELEASE);
```

```
115                 }else if(deltaHeading > 0.1){
116                         motorLeft.run(RELEASE);
117                         motorRight.run(FORWARD);
118                 }else{
119                         motorLeft.run(FORWARD);
120                         motorRight.run(FORWARD);
121                 }
122         }
123 }
124
125 void incrementLeft(){
126         leftTotal++;
127 }
128
129 void incrementRight(){
130         rightTotal++;
131 }
```