Dumpsty

P5 Project
Group SW510E16
Software
Aalborg University
21st Dec 2016

Fourth 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: 52
Appendix range: X
Published 21-12-2016

Synopsis:

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.

Thank the people who should be thanked for helping making the project, here.

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*** section.
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 it is, 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 from.*

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.

Process model

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 and hardware chapters, 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.

The process includes emphasis on a small group of individuals with tasks usually undertaken by two group members or more, 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.

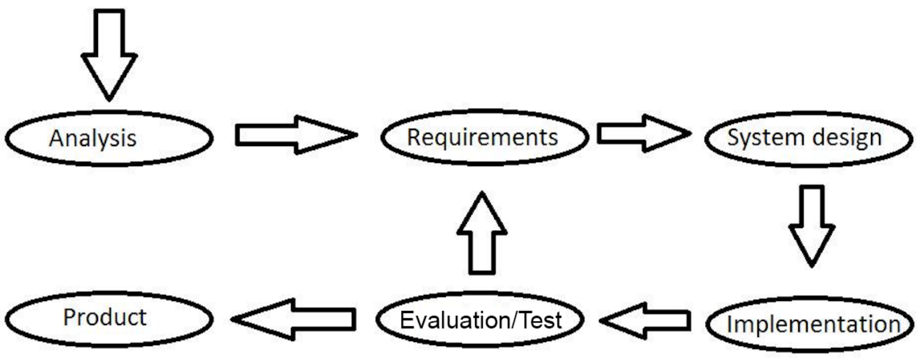

Figure 1. Process model used for the project

Toolchain

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 C#
- Arduino IDE 1.0.3
- BoundT
- UPPAAL

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 C# 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]

Contents

List of Figures

Listings

1 Introduction

An embedded system is a computer system which only have 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 would be designed as a gadget, with a sole purpose of catching trash thrown at it. A trash bin that can catch trash would make the act of cleaning more fun and interactive, and the availability of each individual trash bin would be tremendously increased.

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.

Analysis 2

From the analysis chapter it should be possible to derive the requirements for the smart trash bin, Dumpsty. The initial user stories will help make the user requirements for the project. After the user requirements has been formed, the information from the user stories 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.

2.1 User story

As mentioned before the user stories should help make the user requirements for the project, one user story have been made that showcase the use of Dumpsty.

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 bin instead. His aim when throwing the trash isn't that of a trained basketball player, so he often has to pick up the trash after throwing it at the 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 bin, and the 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 bin from a designated side of a sensory camera, the robotic trash bin should identify the trash, move the 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 catching 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.

2.2 User requirements

The user requirements of the project have been conducted from the section 2.1. 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.

2.3 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, where in this project a table tennis ball will be used. The ball will be bounced on the ground, to give the sensor and robot more time to detect, track and calculate a point, at which the robot should move to, in order to catch the object.

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. This will be used to calculate the point of impact between the object and the ground.

Catching
The catching phase is in which 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.

2.4 Hardware

The following sections will describe the hardware considerations for the project, the hardware's capabilities and it's limitations. The limitations and capabilities will be in consideration when reviewing the requirements later in this chapter, to specify how the hardware can meet the requirements.

2.4.1 Sensors

Sensors were used to measure the environment around the arduino and enable the arduino to understand that environment. The sensors utilized are described in individual sections.

Both the LEGO NXT Gyroscope and Accelerometer were used in the Appendix A, but were chosen not to be used in the evaluation of the increment. Even though they were not used for the final product, they are lightly explained in this section.

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 3 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 concoction be used to position the robot.

2.4.2 LEGO NXT Servo motor

For the robot to be able to move, it would 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 down to a $1°$ 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.

2.4.3 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.cc, a]

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 WifiShield was acquired to enable wifi communication between the Kinect sensor and the Arduino Mega. The intention was 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 a minute. [Arduino.cc, c]

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 would 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 is attached to the shield. [Arduino.cc, b]

2.5 Problem statement

Based on the above analysis and the limitations of the project, a problem statement for the project has been constructed:

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

Design specification 3

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

3.1 System requirements

The requirements for the project have been assembled through out several chapters. The Analysis chapter 2, is were the most of the requirements are conducted. The general requirements were extracted from the user story in section 2.1. Then in the Hardware section 2.4, the hardware would set some limitations for the project which lead to some more detailed requirements in the form of subrequirements.

Through out the work with the robot problems occurred and the requirements therefore changed. The changes done to the requirements though out the project, can be seen in the increments found in the Appendixes A, B and C, where the requirements for the report will be the in the Evaluation section C.4, in appendix C.

The requirements for the project have been conducted through out the whole process and changes were made accordingly, to adapt to the hardware's limitations and the problems found. The requirements can be seen in the following list:

- 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 motory 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 know where the the thrown trash will land
  - 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

3.2 Delimitations

The group have made two delimitations for the project. First the group have decided that multiple object thrown should not be considered, so only one object should be thrown at a time.
It have also been decided that the robot should not drive backwards. The robot starts outside its predefined area and will only drive forward into that. 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 will not be able to drive to the new point given by the Kinect.

4 System design

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.

4.1 Software choices

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

4.1.1 Arduino IDE

After testing the Arduino IDE 1.6.12 with the code defined on the Arduino website's Wifi-guide [Arduino.cc, d], it was discovered that the Wifi library was not compatible in the IDE in a series of newer versions. The library was supported in version 1.0.3, which is why this particular IDE version is used in this project. The newer IDE's has increased security, which would actively refuse any TCP connection from the computer.

4.1.2 Libraries

To be able to use the Motor- and Wifi shield, two libraries had to be included to the project. This gave some new functions, be to able to make the program.

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 have set different speeds and can be stopped at different times. If the library was not included in this project, the group had to write their own code to control the motors and reading the motor encoders.

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 4.1.1 explains complications between the versions of the Arduino IDE, and why the Arduino IDE 1.0.3 is used for this project.

4.2 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 4.1.

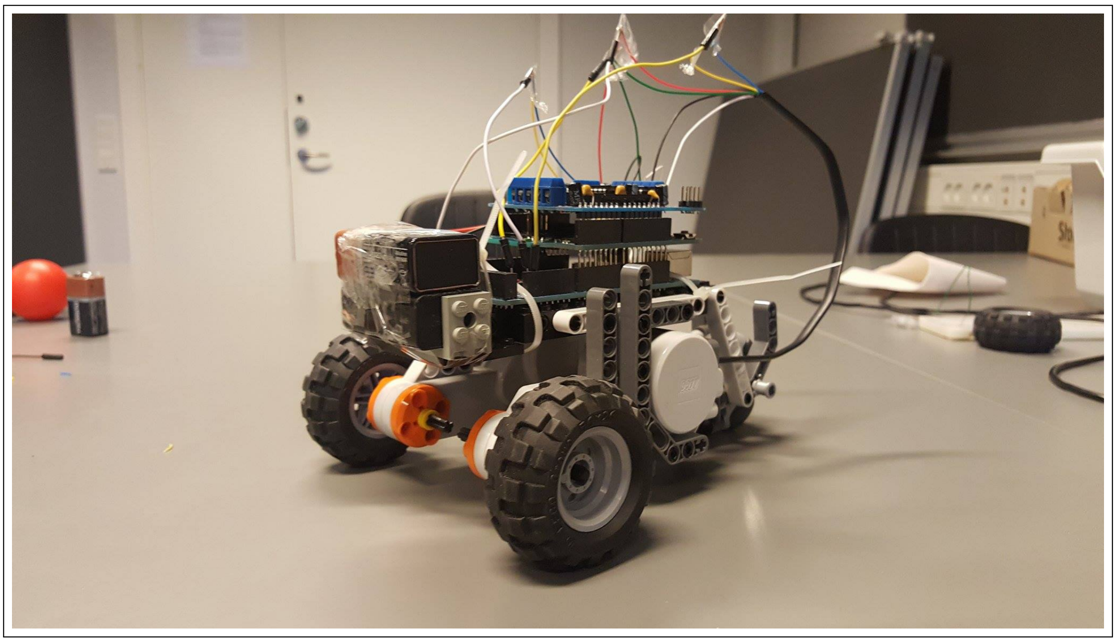

Figure 4.1. The final design of the robot

The robot uses two LEGO NXT Servo motors with corresponding 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, having a kind of platform at the top. The Arduino is stripped 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 was build with the arduino at the top, for easy access to the different pins if needed.

4.3 Arduino program design

As mentioned in section 4.1, the arduino program was developed in the Arduino IDE version 1.0.3. The program will be responsible to communicate to the robot where to move, to reach the collision point, sent by the Kinect of the object thrown at its predefined area. This is done by the program translating the coordinates from the Kinect, into coordinates know for the robot and move to that specific location

The program have a setup and will loop the behavior of the robot. First the setup function, sets up the WiFi connection to the Kinect program, it then waits for the Kinect program to send coordinates for the impact point of the object thrown. When it has received a set of coordinates, it will then translate the coordinates to so it knows where to move to. The next steps, the arduino program will continuously loop through until the program are exited: First it will check if it is at the impact point, if it is, the program is done. Else the robot will start moving, while it keeps adjusting it direction relative to the robots heading. Finally in the loop it will update the robots current position and its heading. All this can be broken down into tasks:

- Make connection to the Kinect
- Wait to receive coordinates from the Kinect
- When the coordinates are received, it will translate these to understandable coordinates for the arduino and then enter the behavioral loop function:
  - Check if already at the collision point, if it is exit program
  - Start moving, adjust direction relative to heading
  - Update its position

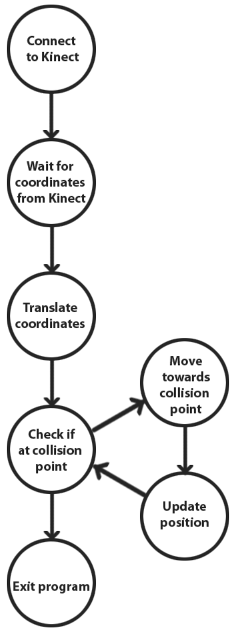

Figure 4.2. The flowgraph of the program

4.4 Robot positioning and movement

The idea for moving the robot would be to send coordinates to the robot, which would be 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 robots 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.3.5, the robots maximum speed was 25.5 cm/s, but it was significantly slower moving backwards than forwards. 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 was determined to only be in front of the robot, to ensure that the robot does not move backwards.

4.5 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 IDE.
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 robots movement. The computer should be able to send coordinates more than once, since the first impact point is not necessarily be 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 robots 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.

4.6 Scheduling

The subsection Cyclic Executive details the model used in this project to schedule tasks and discussion will conclude in the subsection real-time analysis with 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]

4.6.1 Cyclic Executive

A cyclic executive is a model which assumes a fixed set of periodic tasks[Bertolotti og Manduchi, 2012]. 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[Bertolotti og Manduchi, 2012].
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 since there is no scheduling at runtime[Bertolotti og Manduchi, 2012]. Cyclic executives can be constructed separately from the system, by hand or by a tool.

4.6.2 Interrupts and their definition

To talk about interrupts we first need to define what we are talking about, these definitions are quotes from the John Regehr's article [Regehr, 2006]. 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 often but not always return control flow to where, the interrupt disrupted control flow[Regehr, 2006]. 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.
A 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 the interrupt is currently pending[Regehr, 2006]. Most hardware platforms. including the atmega2560, use a bit to distinguish whether an interrupt is pending or not. 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[Regehr, 2006].

4.6.3 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[Regehr, 2006]. 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[Regehr, 2006]. 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 shorter branch at some point during the execution of the program.
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.
Stack overflow is a general problem in embedded systems, but will not be a problem for this project. A common factor that leads to this, is that threads will have allocated memory and in a system with many interrupts, the interrupts can cause the memory which was allocated to the thread to overflow. The arduino mega2560 board used in this project is not using multiple threads, have relatively large memory and does not use more than one type of interrupt which cannot preempt itself.

4.6.4 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[Regehr, 2006]. 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 the interrupt overload can occur, it is possible to analyze how much cpu utilization can be spent handling interrupts. If the cpu utilization for handling interrupts is above 100% then the cpu might not be able to handle all interrupts, neglecting the rest of the system entirely. If the cpu utilization is below 100%, the rest of the system also needs cpu utilization which means the net utilization could become greater than 100% resulting in interrupt overloading starving other functions.
To prevent an interrupt overload from starving other functions, 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 in interrupt handler in our system is 40 clock cycles 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 relatively low amount of cpu utilization is used, this type of problem doesn't occur.

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 4.6.1, which handles many of the problems addressed, as long as its assumptions are true. Since the cyclic executive has predetermined behaviour it will have low 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, low jitter, prevents race-conditions and deadlock(not sure if true, seems true). The very 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 5

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.

5.1 Microsoft Kinect

5.2 Robot

This section will describe arduino code for the robot. This will include how the arduino connects to the Kinect and how behaviour and calculations done by the arduino.

5.3 Scheduling

To calculate the worst-case execution time(WCET) for the systems tasks, several methods were tried out, which is described in Appendix C in section C.2.2. The method used in this project to get the WCET for the system tasks, is to use the micros() function in the arduino IDE.
Each task running time is measured with the micros() function, by running the function several time at using the highest time value as WCET. This was done for the functions updatePosAndHead(), driveTowardsGoal() and for when the Kinect program sends data till the arduino have received it via WiFi.

The results of the WCET of the tasks using the micros() function:

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

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

Tests 6

6.1 UPPAAL schedulability analysis

As described in Appendix C.3.2, Dumpsty contains three tasks: PrA, PrB and PrC. These three tasks all have individual WCET, which is not calculated, but rather tested, since calculating these through assembly proved to be impossible due to unbound loops in libraries. After testing the individual task's 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: 1067 microseconds UPPAAL: 2 milliseconds
- PrB Tested: 732 microseconds UPPAAL: 1 milliseconds
- PrC Tested: 8469 microseconds UPPAAL: 9 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 6.1 depicts the automatas 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 automatas 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);

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 is the same, since this is the minimal interarrival-time(MIT) for coordinates from the Kinect sensor. All code has to be executed within the MIT of these coordinates, since a new coordinate will alter the path the robot choose.

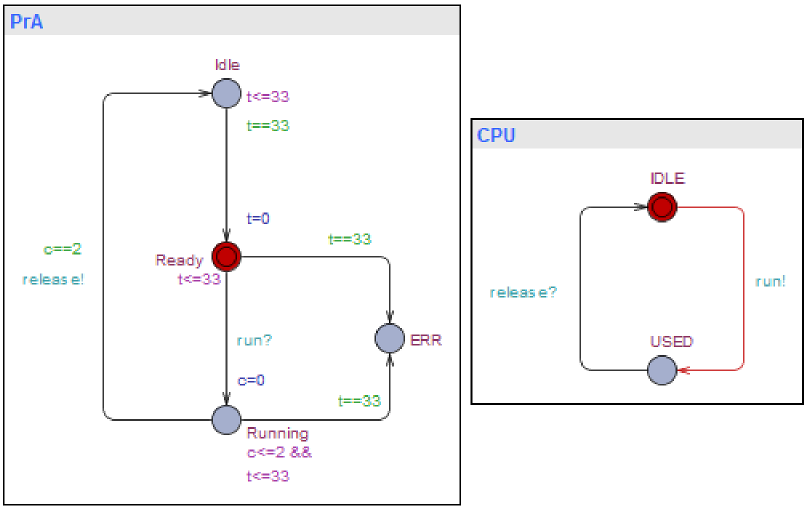

Figure 6.1. Automata in UPPAAL

The last step in the UPPAAL schedulability analysis is to use the verifier. This is done with two queries, found in listing 6.1. 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 6.1. Queries for UPPAAL

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

6.2 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 have been conducted 12 unit tests, which have been split up into two different kind of unit tests, to test the two functions driveTowardsGoal() and updatePosAndHead().
The coordinates sent to Dumpsty in the unit tests are simulated, since this is the best representation of how this 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 described the heading is 0. For a code near prospective of this unit test type, check listing 6.2.

Listing 6.2. First type of Unit test

```
test(curPos30\_150goalPos30\_150Head1){
        heading = itok(1);
        posX = itok(30);
        posY = itok(150);
```

```
        goalX = itok(30);
        goalY = itok(150);
        String result = driveTowardsGoal();
        assertEqual(result, "goal");
}
```

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 6.3.

Listing 6.3. Second type of Unit test

```
test(pos300_250Head075Left12Right5){
        heading = ftok(0.75);
        posX = itok(300);
        posY = itok(250);
        rightTotal = 5;
        rightTemp = 0;
        leftTotal = 12;
        leftTemp = 0;

        updatePosAndHead();

        assertMore(posX, ftok(302.826));
        assertLess(posX, ftok(302.836));

        assertMore(posY, ftok(253.034));
        assertLess(posY, ftok(253.044));

        assertMore(heading, ftok(0.717));
        assertLess(heading, ftok(0.727));
}
```

Table 6.1 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.

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

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 6.1. Table of conducted unit tests

7 Discussion

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 assembler code, but yet again the libraries proved to be a problem, since 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.

Process analysis

The process model used for this project, which is described in section , is a reflection on how our perspective on the processes we usually follow. 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. Catching regular trash is not a hard real-time, safety critical task, unless the trash is a threat to human life or mission critical.
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.
Even though a lot of freedom might be a good thing, it can also be a bad thing for a process model. When working with the process model, it might not have been a good way to structure the work and it has been very experimental at times.
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.

Hardware choices

Object detection

Delimitations

For this project we made two delimitations to the project, which can be found in section 3.2. As mentioned, only one object will be thrown at any given time. This is an environmental delimitation, to not consider multiple projectiles 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 and sending a signal, 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 **??** as the red dot. The robot should then be able to move to both 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.

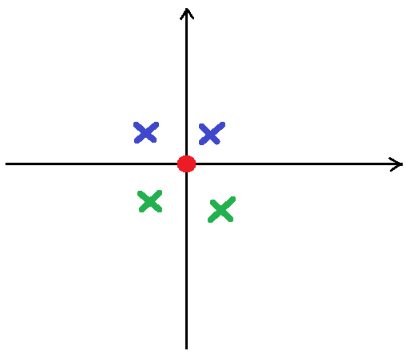

Figure 7.1. Graph describing the coordinate set with no predefined area

8 Conclusion

Based on the analysis chapter concerning a userstory, system capabilities and the hardware's capabilities and limitations, lead to the projects 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 it is not able to catch it, because of the hardware limitations for the project. SKAL DER VÆRE MERE HER?????

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

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

The robot should know where it is positioned

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

The robot should know where the the thrown trash will land

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 receive data from a computer, through a wireless network

The system tasks should be able to be scheduled and verified

Conclusion

Future work 9

Hardware

Bibliography

Arduino.cc, **a**. Arduino.cc. *Arduino Mega 2560.* https://www.arduino.cc/en/Main/ArduinoBoardMega2560. Accessed: 25-09-2016.

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

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

Arduino.cc, **d**. Arduino.cc. *WifiGuide.* https://www.arduino.cc/en/Guide/ArduinoWiFiShield.

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.* 2006.

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

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/.

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 $9.81 m/s^2$, 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* $(9.81m/s^2)$
$\theta:$ *the angle at launchv* $:$ *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.

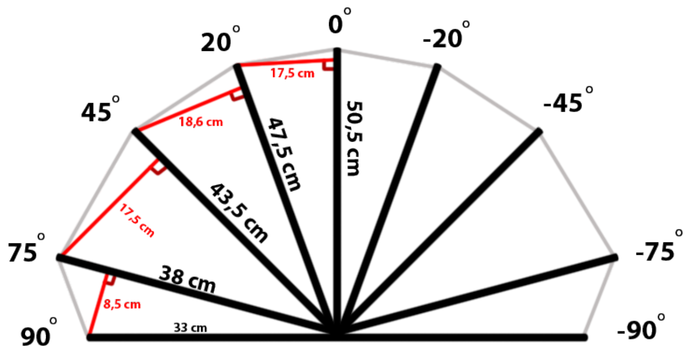

Figure A.1. The robots predefined area

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 2.4.2 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 Microsoft Kinect

A.3.5 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 know where the the thrown trash will land
  - 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 know where the the thrown trash will land
  - 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 2.4.3, 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

```
void setup() {
	Serial.begin(9600);

	Serial.println("Attempting to connect to WPA network...");
	status = WiFi.begin(ssid, pass);

	if ( status != WL_CONNECTED) {
		Serial.println("Couldn't get a wifi connection");
		while(true);
	}
	else {
		server.begin();
		Serial.println("Connected to network");
	}
	ip = WiFi.localIP();
	Serial.println(ip);
}
```

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

```
tcpClient.Connect("192.168.0.100", 9999);
string data = "test";
const int IntSize = 4;
byte[] bytedata;
Stream stream = tcpClient.GetStream();

stream.Write(Encoding.UTF8.GetBytes(data.ToCharArray()), 0, data.Length);
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

```
client = server.available();
while (client.connected()){
	if (client.available()) {
		char c = client.read();
		readString += c;
	}
}
if(readString != ""){
	Serial.println(readString);
	readString = "";
}
```

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

```
volatile int leftTotal = 0;
volatile int rightTotal = 0;
bool motorLeftRun = false;
bool motorRightRun = false;
double heading = 0.0, posX = 0.0, posY = 0.0;
unsigned int tid;
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

```
pinMode(LEFTENCODERPIN, INPUT);
attachInterrupt(LEFTPINTOINTERRUPT, incrementLeft, CHANGE);
pinMode(RIGHTENCODERPIN, INPUT);
attachInterrupt(RIGHTPINTOINTERRUPT, incrementRight, CHANGE);

motorLeft.setSpeed(255);
motorRight.setSpeed(255);

motorLeft.run(RELEASE);
motorLeftRun = false;
motorRight.run(RELEASE);
motorRightRun = false;

leftTemp = leftTotal;
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 know where the the thrown trash will land
  - 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 know where the the thrown trash will land
  - 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 4.5. 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

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 C.1 and C.2 a function written with the AVRFIX library and a function written normally for arduino can be compared. Be aware that the function in listing C.2 might not look exactly like this in the final code for the project.

Listing C.1. The function updatePosAndHead with AWRFIX library

```
void updatePosAndHead(){
int currentLeft = leftTotal;
int currentRight = rightTotal;
fix_t distPrDeg = ftok(DISTPRDEGREE);
fix_t dltL = itok(currentLeft - leftTemp);
fix_t dltR = itok(currentRight - rightTemp);
fix_t deltaLeft = mulk(dltL, distPrDeg);
fix_t deltaRight = mulk(dltR, distPrDeg);
leftTemp = currentLeft;
rightTemp = currentRight;
fix_t deltaSum = deltaLeft + deltaRight;
fix_t dist = divk(deltaSum,ftok(2.0));
fix_t sinHeading = sink(heading);
posX += mulk(dist, sinHeading);
fix_t cosHeading = cosk(heading);
posY += mulk(dist, cosHeading);
fix_t rel = divk((deltaRight - deltaLeft),ftok(WHEELDIST));
heading += atank(rel);
}
```

Listing C.2. The function updatePosAndHead from the Arduino IDE

```
void updatePosAndHead(){
int currentLeft = leftTotal;
int currentRight = rightTotal;
double deltaLeft = (currentLeft - leftTemp) * DISTPRDEGREE;
double deltaRight = (currentRight - rightTemp) * DISTPRDEGREE;
leftTemp = currentLeft;
rightTemp = currentRight;
double dist = (deltaLeft + deltaRight) / 2.0;
posX += (dist * sin(heading));
posY += (dist * cos(heading));
heading += (atan((deltaRight - deltaLeft) / WHEELDIST));
}
```

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 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

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 three tasks for Dumpsty: Updating the position and heading (referred to as "PrA"), driving towards the goal position ("PrB") and the WiFi code that has to be looped ("PrC"). 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

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:

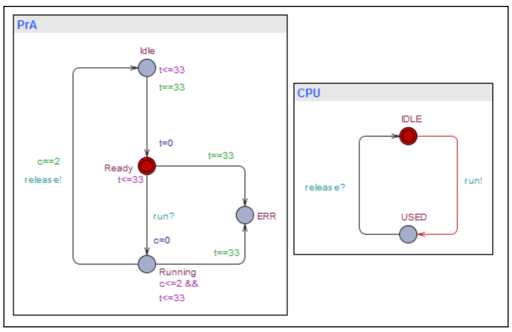

Figure C.1. Automata in UPPAAL

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);

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

```
E<> PrA.Ready and PrB.Ready and PrC.Ready and PrA.t==0 and PrB.t==0 and PrC.t==0 and time>0
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.

EVALUERE PÅ CONNECTING ARDUINO AND KINECT!!!!!!!!!!!!!!

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 know where the the thrown trash will land
  - 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)*)*