Muttley and the Pigeon

An embedded real-time system for
tracking and following vehicles

Student report

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East
http://www.cs.aau.dk

Title:

Muttley and the Pigeon - An embedded real-time system for tracking and following vehicles

Theme:

Embedded Systems

Project Term:
P5, fall 2011

Project Group:
sw504e11

Students:
Anders Eiler
Esben Pilgaard Møller
Thomas Kobber Panum
Magnus Stubman Reichenauer
Rasmus Steiniche
Bjarke Hesthaven Søndergaard

Supervisor:
Ricardo Gomes Lage

Copies: 8

Pages: 119

Finished: December 19th, 2011.

Synopsis:

The Muttley and the Pigeon project involves two cars created with the use of the LEGO NXT and implemented using the nxtOSEK platform. These two cars, where one is the leader and one is the follower, are designed to follow each other as if it was a road-train on a highway. The leader uses the sonar sensors to avoid obstacles, while the follower uses them to keep track of the leader.

Anders Eiler

Esben Pilgaard Møller

Thomas Kobber Panum

Magnus Stubman Reichenauer

Rasmus Steiniche

Bjarke Hesthaven Søndergaard

Preface

We would like to thank our supervisor Ricardo Gomes Lage and René Rydhof Hansen for helping and supervising throughout the project. Furthermore we would like to thank the people of HiTechnic for the support with their hardware.

The following report details the process of creating a road-train with LEGO cars using the LEGO NXT.

In the report the following notations will be used:

The leader car will be referred to as the Runner, and the follower car will be referred to as the Stalker.

Quotations are the words of another person along with a source. The source of the citation will either be in the text immediately before or after, or could potentially be incorporated into the quote as shown in the example below:

> “ This is an example of a quotation. ”
>
> — X, p. Y

References are references to sections, figures, code snippets, chapters or parts written elsewhere in the report. This could look like the following:

This is explained in Section X.Y.

Code examples are written in a special environment so they are easy to read and recognize. Examples can be seen in Code snippet 1. Whenever there is a sequence of three dots (“...”) in a code snippet, it means that we have omitted some content, which is not important in that specific context.

```
1 #include<stdio.h>
2
3 main()
4 {
5     printf("Hello World");
6 }
```

Code snippet 1. Code example of a hello world program written in C.

Contents

Part I

Introduction

Prologue 1

Systems, known as real-time systems, RTS, which react to environmental stimuli, including passage of time, are important in today's world. It has been estimated that 99% of all produced microprocessors are used in embedded systems [20]. We define embedded systems as systems designed to perform a few dedicated functions, with probabilities of real-time aspects. This could for instance be opening the slide doors at a supermarket, measuring the temperature in a given room, or some other trivial function.

This report explains the process of constructing an embedded system by describing the theory involved in RTS and implement an embedded system using the LEGO NXT.

1.1 Motivation

We want to create a system which not only solves a specific problem, but also has the potential of having social influence.

The general idea behind this system comes from the principle of a *road-train*, originally described in [1] for use on highways. The road-train consists of a leader car which is driven by a professional driver, followed by a number of cars that automatically follows the leader.

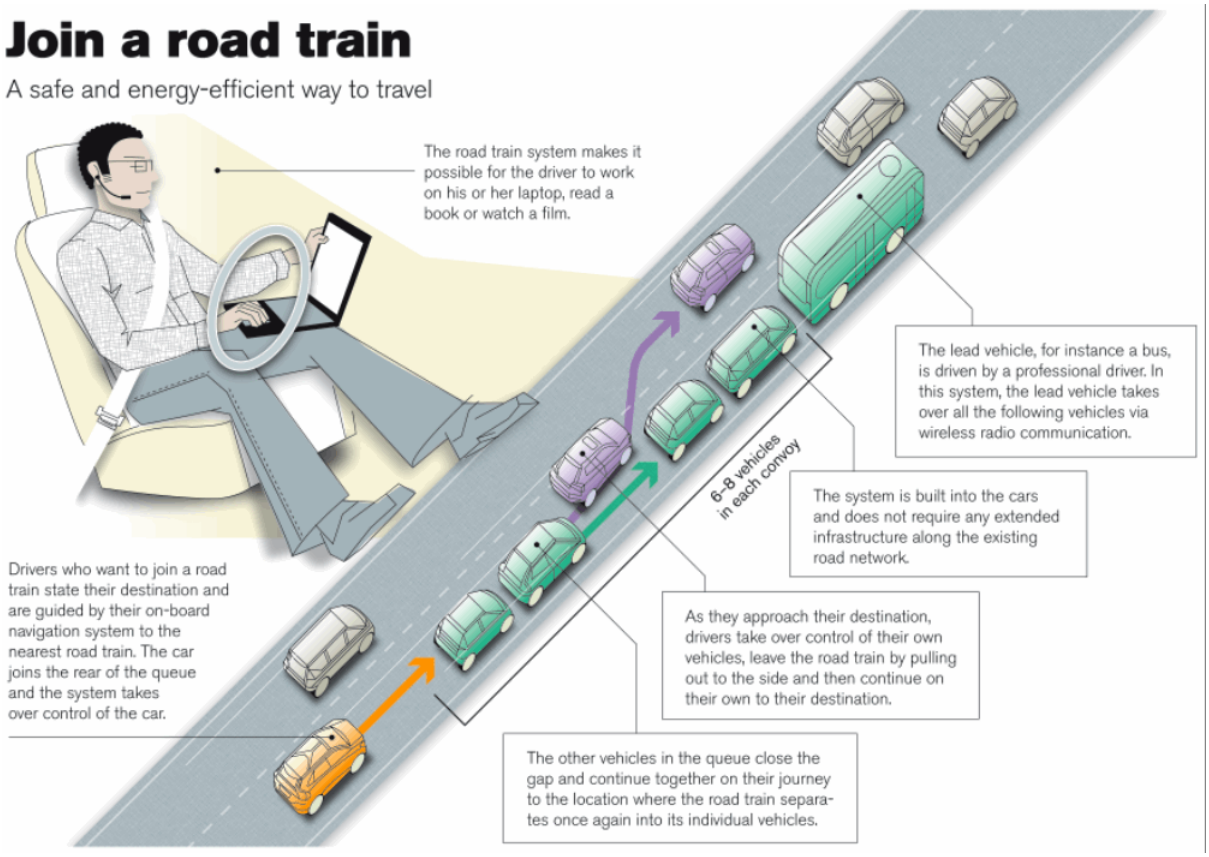

Figure 1.1. Display of the basic ideas behind a road-train (source: [1]).

The leader car can be detected by a sensor on the follower car. When a leader car is detected by a regular car, the regular car can drive up behind the leader car and set itself to follow. When the car has connected to the leader, some software takes control of the car and the driver can relax and watch the car drive on safely.

There are a number of advantages to this technology, for instance: Fewer traffic accidents as the whole driving experience is controlled by a computer, and that the driver can concentrate on working instead of driving (taking advantage of what would otherwise be a waste of time). If the cars are controlled automatically, they can drive closer to each other. This will reduce air resistance, causing the cars to use less fuel and therefore emit less CO2 as they drive. This was shown on an episode of the program Mythbusters on the Discovery Channel [8].

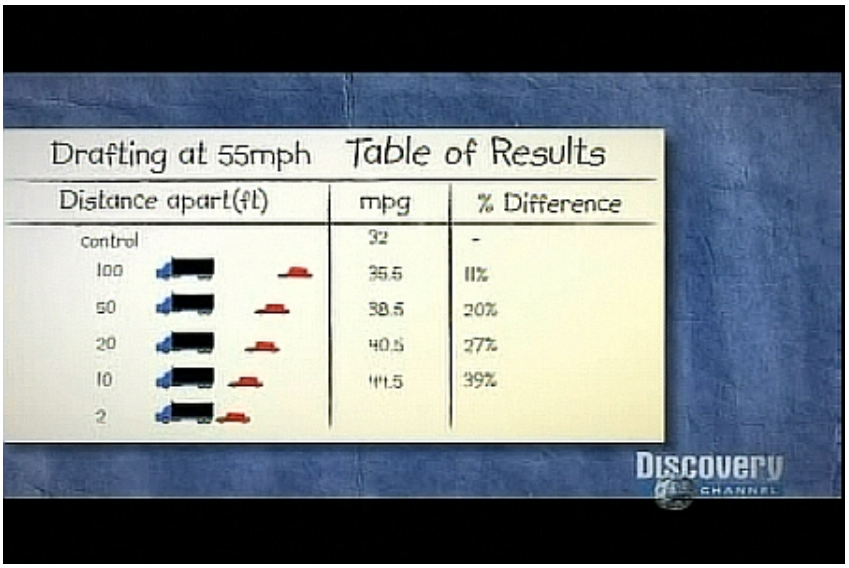

Figure 1.2. A screenshot from the mentioned episode of Mythbusters, showing their air-resistance results (source: [8]).

Showing that traveling 30 m behind the truck increased overall Mile Per Gallon (mpg) efficiency by 11%. Traveling 3 m behind the truck produced a 39% gain in efficiency. The latter could be unsafe to do in real life, if the car is being controlled manually. Therefore a system would have to be implemented to control the distance.

This all together makes up the motivation behind the project.

1.2 Problem Definition

In Section 1.1 the benefits of the road-train technology were explained. The technology is intriguing and has the potential to significantly improve everyday life. Therefore, we want to examime it further. This gives us the following preliminary problem statement:

A lot of time is being used in transport, time that costs money as it prevents the people being transported from working during this time.

One way of solving this problem could be by implementing a road-train system. This would enable people to work instead of drive. There are many real-time aspects to consider. As the project will aim to emulate the described real-life scenario, the same demands that would exist in the real world exist for this project.

The properties of the system are:

- The leader car must lead the follower car safely to its destination
- The follow car must not crash into the leader car
- The follow car must follow the leader cars direction
- The follow car must stay close to the leader car so it does not disrupt the rest of the traffic

1.2.1 Limitations

As this is an educational report, and a limited amount of time and resources are available, the system will also be limited. The resources which we have been granted consist of a small number of LEGO NXT bricks, and additional LEGO pieces. We cannot build real cars and perform tests on an actual highway.

Due to the aforementioned restrictions, the project will be limited to creating a system that has one leading vehicle and one following vehicle, while still conforming to the predefined properties from Section 1.2.1.

1.2.2 Research question

Based on the preliminary problem statement from Section 1.2, its description and the limitations from section 1.2.1, the following research question has been defined:

How can we design and implement a road-train system consisting of one leader vehicle and one follower vehicle, using LEGO NXT?

Real-time Systems 2

This chapter will explain the theory of real-time systems used as a knowledge base for the project. It starts by defining the term real-time systems, followed by some examples and then more detailed theory on the primary topics of a real-time system.

2.1 Definitions

The following real-time systems definition will be used in the project:

> " ... a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment. "
>
> — [20, p. 2]

The key aspect of real-time systems is that deadlines often are critical. If not held the real-time system cannot react to the stimuli from the environment in the right time interval and might not deliver its service. As a consequence of this real-time systems have harder deadlines than normal computer systems, which is the main difference.

In real-time systems, this stimuli is usually called *events*. These events can be one of three kinds: [20, p. 3]

- Periodic
- Aperiodic
- Sporadic

A **periodic** event is one that releases at a set interval. Here, the event is passage of time. An **aperiodic** event is one that releases at the occurrence of stimuli which is not the passage of time. This could be an interrupt or another signal from the environment. A **sporadic** event is identical to an aperiodic event, except that the sporadic event has a bound on how often it can occur in any time interval.

The consequence of an event firing is a *task* release. A task is said to be either periodic, aperiodic, or sporadic, depending on the nature of the event which releases it for execution.

Furthermore, tasks are also said to be *event-driven* if released for execution by an aperiodic or sporadic event, and *time-driven* if released by a periodic event, namely passage of time. The amount of time a task takes to execute is usually bounded by a requirement of how long it may take. This requirement is known as a *deadline*. Whenever a deadline is missed, there are usually consequences. Depending on the nature of the tasks, the fatality of the consequences can be somewhat granular. This granularity is usually split up into three categories of deadlines: [20, p. 2]

- Hard real-time
- Firm real-time
- Soft real-time

Hard real-time systems, are systems where the consequence of missing a deadline will be fatal or unacceptable. **Firm** real-time systems, are systems where the consequence of missing a deadline will result in no benefit of the execution. In other words, missing a firm deadline will not be fatal, as the system will still be able to function. **Soft** real-time systems, are systems where the consequence of missing a deadline is non-fatal or acceptable, and the execution is still somewhat beneficial.

2.1.1 Example

A real-time system, with responsibility to avoid collisions with vehicles in front of it, would typically have two tasks.

The first task would periodically collect stimuli from the physical environment. In this case, the distance to a possible vehicle. If the distance is too small, then a signal could be used to release the second task for execution, which will initiate the braking process in an appropriate manner, and avoid collision. The *signal* would in this case be an aperiodic event.

The consequence of the first task taking too long to finish would result in the second task being signaled too late. Therefore, the braking process might not be started in time to avoid a collision. The consequence of the second task taking too long to execute would have the same effect. Therefore deadlines would have to be set for both tasks, and measurements would have to be made in order to never miss the deadlines. Depending on the purpose of the vehicle, it could be considered as a hard real-time system, since missing a deadline might be fatal.

Any periodic task which performs a physical mechanical action, will be **sporadic**, since it will take some amount of time to perform the physical mechanical action, and therefore, the task cannot be released for execution because it would already be running.

2.2 Reliability and Fault Tolerance

This section will explain basic concepts of reliability and fault tolerance in real-time systems. It is based upon the texts from [20, p. 27] and [3].

2.2.1 Dependability

Dependability is a broad class containing a number of attributes, which are used to describe the overall dependability of a software system. The following definition is used:

> “ The dependability of a system is the extent to which reliance can justifiably be placed on the service which it delivers to its users. It comprises attributes such as reliability, safety and security which relate to different aspects of the users dependence. ”
>
> — [15, p. 29]

The main subclasses of dependability can be seen in Figure 2.1.

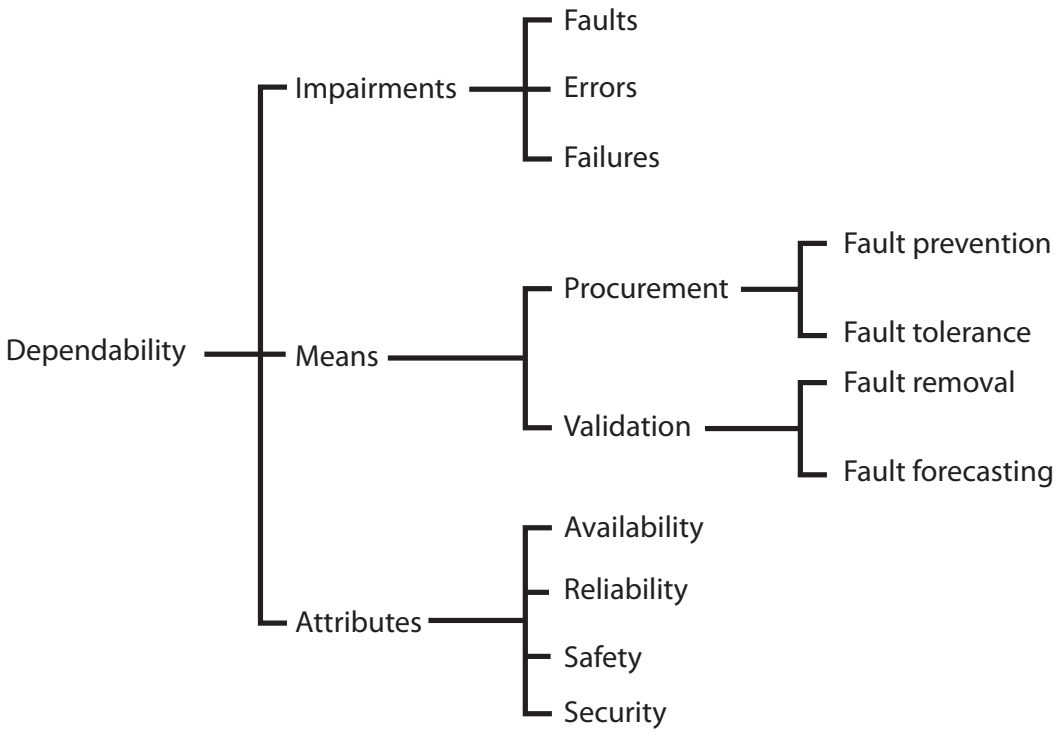

Figure 2.1. Taxonomy of dependability (source: [3]).

When describing dependability in a real-time system it is usually done in regards to the three main subclasses:

- Impairments — These are faults, errors and failures, which can be described as undesired, unexpected, circumstances causing or resulting from nonexistent dependability.
- Means — Techniques to secure the delivery of a service in the right time.
- Attributes — These are used to express the expected properties of a system.

Achieving dependability in a real-time system involves considering the following aspects:

1. Fault prevention — Trying to prevent fault occurrences in the construction phase.
2. Fault tolerance — Trying to tolerate and handle the faults that can happen in a system, making it possible to continue working despite faults being present.
3. Fault removal — Minimizing faults from both hardware and software.
4. Fault forecast — Estimating the occurrence and manifestation of faults about to happen in the system.

The first two, fault prevention and fault tolerance, are methodologies used to construct a dependable system. The last two, are methodologies used to validate a system's dependability.

2.2.2 Reliability

Reliability and fault tolerance have four different properties in real-time systems.

These are defined [3] as:

- Availability — The extent to which a system has a readiness for usage.
- Reliability — The extent to which system continuously provides its service.
- Safety — The extent to which a system avoids catastrophic consequences on the environment.
- Security — The extent to which a system prevents unauthorized access and/or handling of information.

Reliability of a system is defined by [7] to be:

> " A measure of the success with which the system conforms to some authoritative specification of its behavior. "

From what Randel [7] defines reliability of a system to be, it is also possible to define what a system failure is, again quoting from [7]:

> " When the behaviour of a system deviates from that which is specified for it, this is called a failure. "

Reliability and safety are often important in real-time systems, as it may not always be possible to restart the system if a fault occurs. A real-time system may need to run continuously as for instance it may be inaccessible due to its location, and therefore cannot be restarted. Instead the real-time system must try to handle faults before it starts producing errors.

2.2.3 Faults, errors & failures

There are three different kinds of impairments to dependability: Faults, errors and failures. They can all be defined by some state or property in the system which is not accepted e.g. the system is deriving from what it was intended to. These states and properties can arrive at any time in a given real-time system. To distinguish between acceptable and unacceptable behavior in the system, a specification of what is considered acceptable behavior in the system must be made. Should the system differ from this behavior, a system failure is said to have occurred. The system failure happened due to some erroneous state, i.e. an error where e.g. the system is in a state which is different from a valid state. The conditions which caused the error are faults.

The nature of faults can be described as accidental faults, faults that are created or appear unintentional and intentional faults (faults that are created on purpose). The origin of faults can be classified as:

- Physical faults — Related to a physical phenomenon
- Human-made faults — Inherited from human actions
- Internal faults — Exists in a system which, when invoked, will produce and error
- External faults — Result from interference or interaction with the environment
- Design faults — Imperfections in either the creation or modification of a system, or the operating procedures of the system.

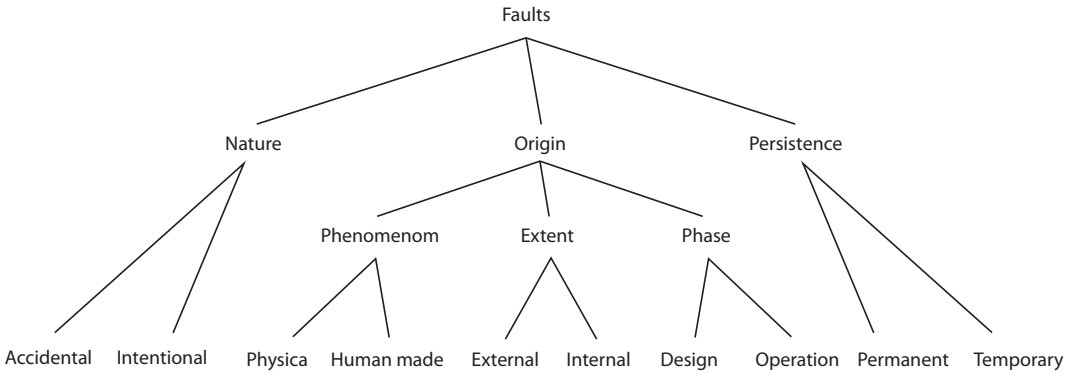

Figure 2.2. The classification of faults (source: [3]).

With the classification of faults it is possible to assign distinguishing properties to software faults. Such faults are accidental human-made permanent internal design faults, i.e. software faults only come from mistakes in design and implementation. This is because the software does not suffer from functional changes from external interactions or aging. This can been seen in Figure 2.2.

An error is a system state which might lead to some subsequent failure, but it depends on a set of factors whether the error produces faults or not. A system that implements some redundancy may mask the error, or the system may overwrite the error e.g. a faulty value or the error may produce a system behavior, which does not seem as a failure from the user's point of view.

A failure is defined as occurring whenever the external behavior of a system does not conform to that prescribed by the specification. From this it is possible to define failure modes, which are categorized according to domain, perception by the user, and consequences.

2.2.4 Failure modes

A system provides a service which can fail in many different ways. Due to the classification of the provided service, it is possible to classify a system's failure modes, according to the impact they have on the delivered services. Failure modes can be divided into two domains:

- Value failure — The value associated with the service is in error
- Time failure — The service is delivered at a wrong time

Both are defined according to the system specification and measured hereby.

Failure in the time domain can result in the service being delivered either too soon (earlier than required), too late (later than required — often a performance error) or infinitely late (never delivered – often called an omission failure).

The failure perception, as seen by the user, can be categorized into two categories:

- Consistent failures — Failures which are perceived in the same way in all environment.
- Inconsistent failures — Failures for which different environment may have different perceptions.

Inconsistent failures are often caused by Byzantine faults. Byzantine faults is a sub-field of fault tolerance, and a Byzantine failure is when a system fails in arbitrary ways: The service does not stop or crash, but processes a request incorrectly, corrupting their local state or producing incorrect or inconsistent output [21].

2.2.5 Faults

To avoid faults, the real-time system in question must be examined to determine if some hardware components, or some pieces of code might produce faults, and then implement methods to handle them. This is not trivial, however, as it requires extensive testing to locate possible sources of faults, and even with extensive testing there is no guarantee that all possible faults are discovered.

Understanding where faults originate from is essential to solving them. Faults originate from four different parts of a real-time system:

1. Faults due to misinterpretations of the environment which the system then react upon.
2. Faults in software components.
3. Faults in hardware components.

4. Faults from transient or permanent interference in the supporting communication subsystem.

Real-time systems have faults, errors, and failures. Software faults are often called bugs. Bugs are usually both hard to isolate and identify. It is possible to see fault, error and failure as a chain reaction as presented in Figure 2.3. It shows a program running in some state represented as "..." until some fault occurs. The fault can become an error which if not handled properly can cause a failure in the system, which might cause more faults to occur.

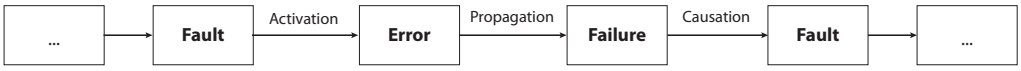

Figure 2.3. Fault, error, failure, fault chain (source: [20])

A fault is said to be active when it produces an error and until that point it is dormant. When an error is produced there is a chance it will transform into additional errors as it propagates through the system. Eventually such an error will manifest itself at the boundaries of the system and cause it to fail.

From a real-time perspective there can be distinguished between three different types of faults:

1. Transient faults occur only at a particular time and then remains in the system for some period before disappearing. Transient faults will be dormant but can become active and produce errors at any time. This type of faults are often hardware related because of interference e.g. some hardware component which is susceptible to interference and when the interference is gone the fault disappears.
2. Permanent faults occurs at a particular time and will remain in the system until they are repaired. This type of faults can both be in hardware or software, i.e. faulty hardware or software design.
3. Intermittent faults are transient faults which only occur from time to time. This type of faults is often seen in hardware, but can easily occur in software as well, as a component e.g. gets over heated, stops working and then starts back up again when it is cooled down.

2.2.6 Fault Tolerance

Fault tolerance techniques are often based on backwards recovery such as rollbacks. It may not always be a satisfying solution in a real-time system because this may take too long or have some properties a real-time system cannot tolerate, e.g. resetting the system to some state before the fault occurred. Another reason why this fault tolerance technique is often unsatisfiable, is the cost of implementing these fault tolerance schemes. The cost of fault tolerance schemes may be prohibitive for many applications.

2.3 Scheduling

Scheduling is a consequence of the activity of restricting the nondeterminism found within concurrent systems and is implemented using a scheduling scheme. A scheduling scheme provides two things: [20]

- An algorithm for ordering the use of system resources e.g. CPU(s).
- A method for predicting the worst-case behavior of the system when the scheduling algorithm is applied.

A scheduling scheme can be either static or dynamic. In a static scheme all priorities are set before execution e.g. the scheduling cannot change at run-time. In a dynamic scheme decisions made at run-time are used in the scheduling. A scheduling scheme involves several components, which depends on the chosen scheme, including priority assignment algorithms and schedulability tests.

2.3.1 Cyclic Executive Approach

If there is a set of purely periodic tasks, i.e. there are no event-driven tasks, it is possible to make a complete schedule in advance, such that the repeated execution of this schedule will cause all tasks to run at their correct rate. The cyclic executive is therefore, essentially, a table of procedure calls. The complete table of procedure calls is known as the major cycle which consists of a number of minor cycles with a fixed duration.

Therefore the length of a major cycle is often equivalent to the longest period of a given task, and it is repeated during execution of the program. A major cycle is divided into a number of fixed-duration minor cycle, e.g. if there are four minor cycles with a duration of 25 ms the major cycles duration would be 100 ms, as $4 * 25ms = 100ms$. During execution of the program, a clock will interrupt for every minor cycle, in this case every 25 ms. This will enable the scheduler to loop through the four minor cycles.

Task	Period, T (ms)	Computation time, C (ms)
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Table 2.1. Simple example of tasks in a cycle execution scheme (source: [20, p. 366]).

In each of the minor cycles, `a` and `b` would be run, which is illustrated in Figure 2.4. In Table 2.3.1 `a` and `b` runs every 25 ms meaning they run in every minor cycle. Here `c` and `d` has a period of 50 ms, meaning `c` can run in the first and third minor cycle, and `d` can run in the second and fourth minor cycle. `e` has a period of 100 ms, meaning `e` can run at any time, but will allow most available time if run in the second or fourth minor cycle.

The properties of the cyclic executive approach are:

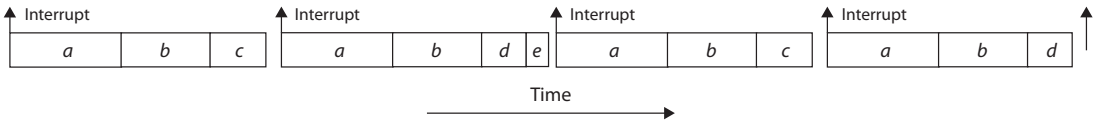

Figure 2.4. Time-line for task set (source: [20])

- No actual tasks exists at run-time — they are merely sequences of procedure calls.
- The procedures share the same address space, meaning they can share data between each other, this shared data does not need protection e.g. via mutexes, as no concurrent access is possible. [20]
- Has the advantage of being fully deterministic.

With the simplicity of this approach comes a number of drawbacks.

- It is difficult to incorporate sporadic tasks or tasks with long periods.
- The major cycle time is the maximum period the scheduling can handle without secondary schedules.

Cyclic execution schemes are using the philosphy *prove it by construction*. If it is possible to construct the scheme, it is schedulable. This is an appropriate implementation strategy for periodic systems, but it is not that extendable. A more dynamic and flexible approach would be a task-based scheduling scheme.

2.3.2 Task-based Scheduling

Task-based scheduling supports task execution directly, as expected by a general-purpose operating system. It determines which task should execute at which times. Using this approach, a task can be in one of three states:

- Runnable
- Suspended, waiting for a timing event, periodic tasks.
- Suspended, waiting for a non-timing event, sporadic tasks.

There are a lot of different scheduling approaches. There are, however, three that are used more often than the rest.

Fixed-Priority Scheduling (FPS) — FPS is the most widely used and also the simplest approach to scheduling [2]. Each task has a fixed and static priority, which is set before run-time. The approach is static. The runnable tasks are executed in the order determined by their priority, which is the individual tasks temporal requirement and not importance to the system. Rate-monotonic assignment is often used in FPS. In rate-monotonic assignment each task is assigned a priority based on its period, if the task has a short period it is assigned a high priority, e.g. the shorter the period, the higher the priority [20, p. 370].

Earliest Deadline First (EDF) Scheduling — In EDF all runnable tasks are executed in the order determined by their absolute deadlines. In essence, the next task to run is the one with the nearest deadline. The absolute deadlines are computed at run-time, therefore this approach is dynamic. To compute the absolute deadlines, the relative deadlines must be known. A relative deadline is one defined as e.g. 25 ms after release, and based on this an absolute deadline can be computed.

Value-Based Scheduling (VBS) — If the current utilization of a system becomes greater than 100%, which might happen if the system has to many tasks, and a utilization test yields a result greater than 100 %. Then the use of static priorities or deadlines becomes insufficient. In these cases a more adaptive approach is needed. This often takes the form of assigning a value to each task and employing an online value-based scheduling algorithm to decide which task to run next.

FPS is supported by various real-time languages and operating system, and can to some extend be considered as the standard.

2.3.3 Preemption and non-preemption

If a low-priority task is running in a priority-based scheduling system, a higher task may be released during this execution. If this happens, two things can occur, depending on whether the system is using preemption or non-preemption.

- **Preemption** — An immediate switch from executing the low-priority to the high-priority task will take place, and the high-priority task will be allowed to execute.
- **Non-preemption** — The lower-priority task will be allowed to complete before any other tasks can execute.

Using preemption allows high-priority tasks to be more responsive, and hence it is preferred. There are, however, alternatives to the two extremes, preemption and non-preemption. Such strategies, that allow a lower-priority task to continue to execute for a bounded time, which means it may not necessarily complete, are known as deferred preemption or cooperative dispatching. These are not relevant for this project and will not be investigated further and are only mentioned to show that methods that lies in between preemption and non-preemption exists.

Starvation or indefinite postponement [20, p. 150] is an error condition in which a task is denied access to a resource it needs, because other tasks are constantly gaining access before it. A task which is being starved may end up failing to deliver its service on time which may lead to a system failure.

2.3.4 Utilization-based Schedulability Testing for FPS

Schedulability, for a task set using FPS with rate-monotonic assignment, can be tested by considering the utilization of the task set. All N tasks will meet their deadlines, if the following condition is true:

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq N \left(2^{1/N} - 1 \right)$$

Here C is the computation time of the task and T is the period of the task. According to [20, Table 11.4] a table for the utilization bounds exists as shown in Table 2.2.

To find the bound for a large number N the logarithm is applied to the right hand side of the before mentioned equation.

$$\begin{aligned} \log(N(2^{1/N} - 1)) &= N * \tfrac{1}{N} * \log(2) - \log(1) \\ &= 1 * (\log(2) - \log(1)) \\ &= \log(2) - 0 \\ &= \log(2) \\ &= 69.3\% \end{aligned}$$

Thus for a large number N the bound will approach 69.3%. Therefore for any N tasks with a combined utilization less than 69.3% it is schedulable, if a preemptive priority-based scheduling scheme is being used. It must be noted however that with 2 tasks the combined utilization can be as high as 82.8%.

The test is not a necessity for a given task set to be schedulable, it is merely a sufficiency test. This means that a task set can be schedulable even though it is not within the given utilization bounds. So to conclude; a given task set can be proved to be schedulable by showing that the utilization test holds for the task set, but if the utilization test does not hold for the task set it cannot be proven as not schedulable.

N	Utilization bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

Table 2.2. Utilization bounds.

2.4 Response Time Analysis (RTA) for FPS

This section is based on [20, section 11] and [16, section 2].

In a single processor system a set of independent tasks τ_k, where k is the general case, executes with $k = 1, 2, ..., m$. Each of the tasks are periodic and all share the following properties:

- A fixed period, T_k.

- A fixed deadline, $D_k \leq T_k$.
- A maximum release jitter, J_k.
- A maximum blocking time, B_k.
- A maximum execution time C_k.

RTA can be used to predict the worst-case response time, **WCRT**, denoted R^w or R. The analyzed task is denoted `task i` while `task j` is in $hp(i)$. $hp(i)$ is the set of tasks with higher-priority than `task i`. WCRT for `task i` can be calculated with Equation 2.1. The WCRT of a given task is compared to the deadline of the task to see if it holds, thus it is required that each task is analyzed individually.

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.1}$$

For the task with the highest priority the WCRT will be equal to its own computation time and maximum blocking, thus $R_i = B_i + C_i$. All other tasks in the scheduling will have interference from higher-priority tasks.

This gives the general case as seen in Equation 2.2, where I_i is 0 for the highest-priority task.

$$R_i = B_i + C_i + I_i \tag{2.2}$$

Here I_i is the maximum interference `task i` can experience at any time in the interval it is run, thus I_i is defined as in Equation 2.3.

$$I_i = \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.3}$$

where I_i is the maximum interference `task i` can experience from a higher-priority `task j`. In this case the following equation is available:

$$N_i = \left\lceil \frac{R_i}{T_j} \right\rceil$$

where N_i is the maximum number of releases that can occur within the interval $[0, R_i[$. Due to the fact that a `task i` can have multiple higher-priority tasks $hp(i)$, it is necessary to calculate I_i for all tasks in $hp(i)$.

The maximum blocking for `task i`, denoted by B_i, is defined as the maximum amount of time a lower-priority task can block it. Maximum blocking depends on the technique used to avoid priority inversion. In this case it is assumed to be OCPP *(Original Ceiling Priority Protocol)*. This means that the maximum blocking is defined by the computation

time C_j of `task j`, where `task j` has the longest computation time of all the tasks with lower priority than `task i`. `task i` can only be blocked once every release, due to the priority inversion used. Therefore this is in most cases an over-approximation as it is assumed that it is the task with the longest computation time that is blocking `task i` such that:

$$B_i = \max_{k=1}^{k} usage(k,i)C(k) \tag{2.4}$$

In Equation 2.1 the exact interference is unknown as R_i is unknown since it is the value being calculated. The equation is difficult to solve due to the ceiling function. Therefore this equation cannot be solved analytically. A typical approach to solving this is by forming a recurrence relationship. This recurrence can be seen in Equation 2.5.

$$w_i^{n+1} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{2.5}$$

Here n is an index for a particular response time. The base case is given as $w_i^0 = B_i + C_i$. When the equation converges, i.e. reaches a fixed point, it is solved. In order to define why a fixed point is the solution to Equation 2.5, some minor theory behind monotonically non-decreasing functions is introduced.

For all x and y, where $x < y$ it is true that:

$$f(x) \leq f(y)$$

If it is monotonically non-decreasing.

This is true for the set: $\left\{w_i^0, w_i^1, w_i^2, ..., w_i^n\right\}$, thus making it monotonically non-decreasing.

This means that because the equation never decreases a solution has been found if $w_i^n = w_i^{n+1}$. If there is no solution the equation will not converge, e.g. no fixed point can be found. A solution will never be found if the utilization of the full set is greater than 100%. Once w_i^n gets bigger than the task period, T_i, it is safe to assume that the task will never meet its deadline, D_i. The starting value for the process, w_i^0, must not be greater than the final, still unknown, solution R_i^w. If this happens the task will exceed its period, which would make the set of tasks not schedulable according to the RTA. Therefore it is important to pick a safe starting value, which could be C_i, since:

$$\begin{aligned} R_i &\geq C_i \\ w_i^0 &= C_i \end{aligned}$$

It is also possible to get an intuition of w_i^n from the problem domain. `task i` has a point of release T_i, until the task completes F_i, the processor will be executing tasks with

priority P_i or $hp(i)$. The processor is at this point said to be executing a P_i **busy period**. w_i can be seen as a time window, which is moving through the busy period. At `time 0` in the time window all $hp(i)$ tasks are assumed to also have been released. This leads to the first value, aside from the start value, for `task i`, denoted w_i^1:

$$w_i^1 = B_i + C_i + \sum_{j \in hp(i)} C_j \tag{2.6}$$

This will end the busy period unless some higher-priority task is released in the meanwhile. If the period is ended, the time window will need to be pushed out further. The window will expand and, as a result, more computation time will get assigned to the window. If this continues the busy period is said to be unbounded, e.g. there is no solution. If a time window, which is expanding, at any time, does not get another *interference-hit* by a higher-priority task it is safe to assume that the busy period has ended, thus the size of the busy period is the WCRT of the task.

2.4.1 Jitter

There is another part of Equation 2.5 which has been left out until now. This is the aforementioned jitter. Jitter is the time it takes from when an action is supposed to happen to it actually happening, and can be specified as release and response-jitter. This depends on the platform. Therefore the WCRT for a `task i` with Jitter is as in Equation 2.7.

$$R_i = w_i + J_i \tag{2.7}$$

where

$$w_i^{n+1} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n + J_j}{T_j} \right\rceil C_j \tag{2.8}$$

2.4.2 Example

This example will show how to make a RTA for a FPS.

Task	Period, T	Computation time, C	Priority, P
a	5	2	3
b	10	3	2
c	40	7	1

Table 2.3. Example task set X.

In Table 2.3, it is assumed to simplify calculations that all the processes in `task set X` is independent, hence there will be no blocking, thus B_i for all tasks will be 0. It is assumed for the sake of simplicity that there is no jitter on the platform, thus making $J_i = 0$ The

highest priority `task a` has a WCRT equal to its computation time, in this case $R_a = 2$. The WCRT for `task b` will be calculated, as it has at least one higher-priority task. Let w_b^0 be equal to `task b's` computation time, $w_b^0 = 3$. w_b^n is calculated until the value converges:

$$\begin{aligned} w_b^0 &= 3 \\ w_b^1 &= 3 + \left\lceil \tfrac{3}{5} \right\rceil 2 = 5 \\ w_b^2 &= 3 + \left\lceil \tfrac{5}{5} \right\rceil 2 = 5 \end{aligned} \tag{2.9}$$

As the value converged, i.e. $w_b^2 = w_b^1 = 5$, the response time of `task b` is $R_b = 5$.

The WCRT for the last `task c` is then calculated:

$$\begin{aligned} w_c^0 &= 7 \\ w_c^1 &= 7 + \left\lceil \tfrac{7}{10} \right\rceil 3 + \left\lceil \tfrac{7}{5} \right\rceil 2 = 17 \\ w_c^2 &= 7 + \left\lceil \tfrac{17}{10} \right\rceil 3 + \left\lceil \tfrac{17}{5} \right\rceil 2 = 24 \\ w_c^3 &= 7 + \left\lceil \tfrac{24}{10} \right\rceil 3 + \left\lceil \tfrac{24}{5} \right\rceil 2 = 29 \\ w_c^4 &= 7 + \left\lceil \tfrac{29}{10} \right\rceil 3 + \left\lceil \tfrac{29}{5} \right\rceil 2 = 31 \\ w_c^5 &= 7 + \left\lceil \tfrac{31}{10} \right\rceil 3 + \left\lceil \tfrac{31}{5} \right\rceil 2 = 36 \\ w_c^6 &= 7 + \left\lceil \tfrac{36}{10} \right\rceil 3 + \left\lceil \tfrac{36}{5} \right\rceil 2 = 38 \\ w_c^7 &= 7 + \left\lceil \tfrac{38}{10} \right\rceil 3 + \left\lceil \tfrac{38}{5} \right\rceil 2 = 38 \end{aligned} \tag{2.10}$$

As `task c` also converged, i.e. $w_c^6 = w_c^7 = 38$, giving it a response time $R_c = 38$. All three tasks hold their deadline, hence $R_a \leq 5$, $R_b \leq 10$ and $R_c \leq 40$.

A RTA has the advantage that it can prove schedulability. It must be noted that RTA can only give a correct result if and only if the computation time estimations, C, are set correctly. If this is the case and the task set passes the RTA it will always meet its deadline. However, if it fails then at run-time, at some point a deadline will be missed.

Task	Period, T	Computation time, C	Priority, P	Worst Case Response Time, R^w
a	5	2	3	3
b	10	3	2	5
c	40	7	1	38

Table 2.4. Task set X.

Part II

Analysis

Hardware 3

Before it is possible to program the real-time system, the hardware components available needs to be tested to determine any margin of error they might present. It is not possible to replace the hardware, so it is important to emphasize any errors it might have, so they can be compensated for in the software.

3.1 LEGO NXT

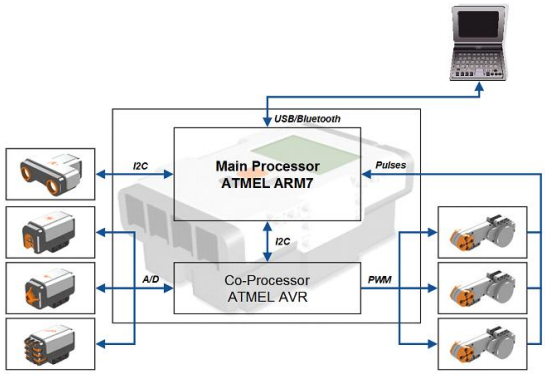

Figure 3.1. LEGO NXT computer.

The LEGO intelligent NXT is the computer used. It can take input from up to four sensors and control up to three motors simultaneously. The NXT has a 100 x 64 pixel monochrome LCD display and four buttons to navigate the various interfaces. It has two processors, a main processor that handles information processing and communicates using the I2C interface and a second processor is used to handle communication with A/D sensors and sends their input to the main processor using the I2C interface. The main processor is an Atmel 32-bit ARM processor with 256 KB FLASH and 64 KB RAM. The co-processor is an Atmel 8-bit AVR processor with 4 KB FLASH and 512 Byte RAM. The NXT is able to communicate via Bluetooth (CSR BlueCoreTM 4 v2.0 +EDR System) or the 12 mbit/s USB2 connection.

3.2 LEGO Motor

The LEGO NXT Servo motor is an essential part of the hardware as it makes both movement and steering of the cars possible. The motor is a 9V motor. In order to program appropriate software, a series of tests were conducted to examine the motors behavior.

To determine the speed of the engines, speed tests were conducted to determine how fast the motor runs. The absolute number of the Pulse-Width Modulation *(PWM)* is the percentage of how much of the motor's power is being used, e.g. 20 PWM (and -20 PWM) is equal to 20%. PWM is used to control the flow of current to an electronic device. The higher the PWM value the more power is given to the device.

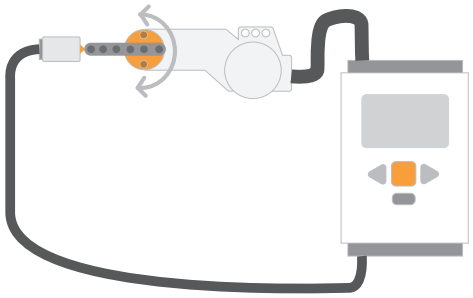

Figure 3.2. LEGO NXT Servo motor test setup.

The following tests were conducted using the setup in Figure 3.2: The motor was programmed to run with a PWM value of 20, 40, 60, 80, and 100. A brick was attached to the axis of the motor, so when the motor passed a certain point the brick pressed a touch center. A program was written, which increments a counter by one when the touch sensor is pressed. The test was timed using a digital clock stopping the motor after 60 seconds.

The test results in Figure 3.3 are subject of user faults, meaning several factors can have had an effect on the results. The results are somewhat linear and shows that the motor is probably designed to run with 160 RPM at full speed.

Figure 3.3 shows the tests for another of the four motors made available for the project. This result is even closer to 160 RPM at full speed. Therefore it will be assumed that all motors run at 160 RPM at full speed. This assumption is made since the hardware design chosen and described in Chapter 5 does not have any requirements for motors being calibrated to have identical speeds at the same PWM.

3.2.1 Precision test

The API for the LEGO servo motors in the nxtOSEK operating system contains a function to read a counter from the motor. The counter shows how many degrees the engine has turned since it was last reset. Default value is 360 degrees to one full turn of the motor.

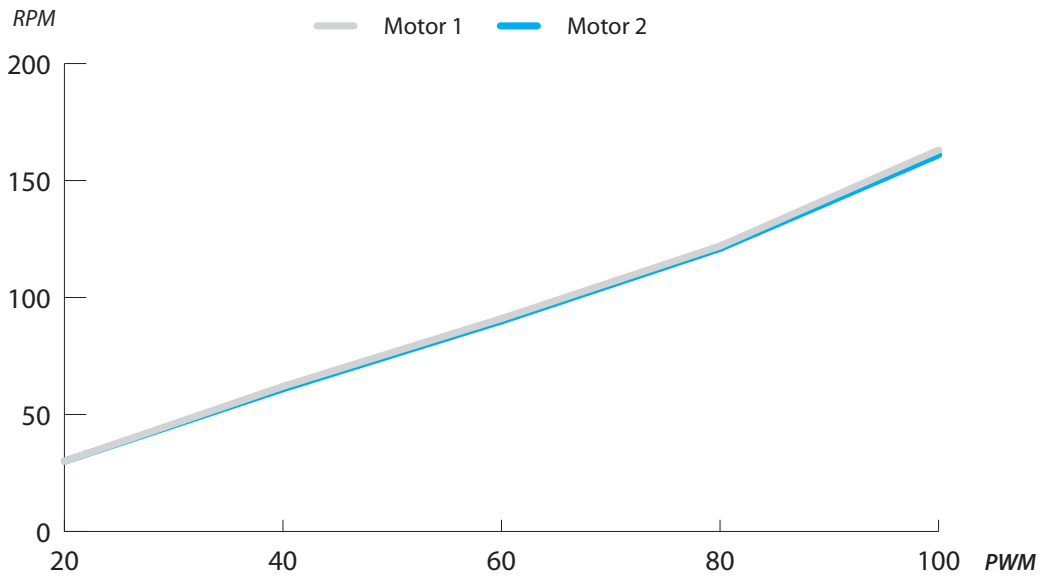

Figure 3.3. Speed test for LEGO NXT Motors.

In order to test this, the NXT was programmed to turn the engine slowly and stop it as soon as it hit 360 on the motor-counter. The same setup was used as in the speed test.

PWM	Count
20	359
40	361
60	360
80	365
100	368

Table 3.1. Precision test for LEGO NXT Motor 1.

It must be taken into account that the engine does not stop immediatly when the `nxt_motor_set_speed(NXT_PORT_A, 0, 1)` command is called. The engine uses a few degrees to stop completely, which is described in Section 6. Taking this into account it is assumed the original goal of 360 counts per motor round stands, and that it can be used as an offset in the system.

3.3 LEGO Sound Sensor

The LEGO sound sensor is a piece of hardware, which enables the LEGO NXT to measure the raw sound level. The sensor enables the possibilities of calculating the distance and direction of a sound source, as explained in Section 4.4. Another use of the sound sensor would be to have the car activate on a sound at a certain dB, and furthermore deactivate when the sound is played again.

A problem with the sound sensor is that it requires that the background noise is limited to a certain dB. If there is a lot of background noise the sensor needs an even louder sound source to recognize it.

3.4 LEGO Sonar Sensor

The LEGO sonar sensor is used for measuring distances by *Sound Navigation and Ranging* (Sonar). The sensor consists of a transmitter and a receiver placed with 3.5 cm between their centers. The transmitter is placed at the right side on the sensor, and the receiver at the left side.

3.4.1 Distance Test

The test was done by placing the sonar sensor at a distance measured by a ruler, the sensor would output to the LCD and the result written down when the output stabilizes.

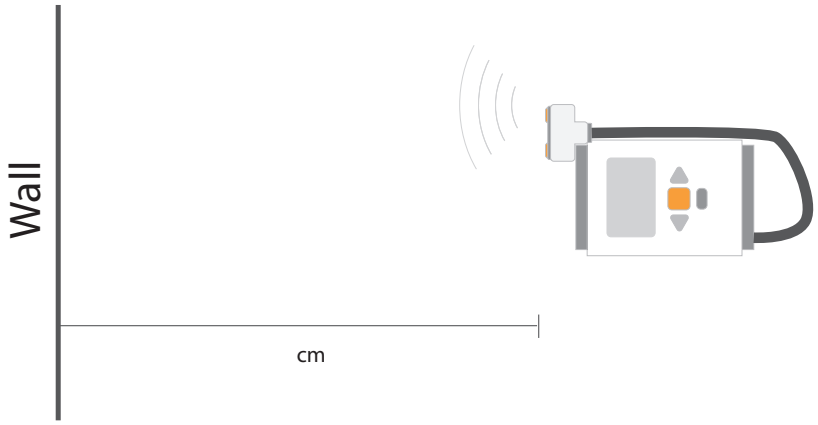

Figure 3.4. LEGO NXT Sonar Sensor Distance Test.

For the test the following distances were measured (in cm): 5, 10, 20, 40, 60, 80, 100, 125, 150, 175, 185 and 200. These measurements were done three times for each sensor, while alternating between the sensors. Five sensors will be tested and named: S1, S2, S3, S4 and S5. The average results for S1 and S4 are listed in Table 3.2 and Table 3.3 respectively. The raw data for S1 and S4 as well as the data for sensor S2, S3, and S5 can be found in Appendix A.3.

The average data for S1 and S4 is shown as they represent the worst and best sensor available respectively.

The average test data from Table 3.2 and 3.3 show a high relative deviation at short distances. This problem is consistent across all five sensors and must be an issue with the hardware type rather than the specific unit.

The data for S1 suggests that it is a very unreliable sensor, whereas S4 is very accurate as soon as the distance is more than 20. Therefore creating a general piece of software for handling input from both S1 and S4 is not possible. Adjustments can be made to the software, however, since it is known which sensors are used and how reliable they are.

Reference Distance	Measured Distance	Relative Deviation
5	7.33	46.67%
10	12.67	26.67%
20	23	15%
40	41	2.5%
60	61	1.67%
80	81	1.25%
100	101	1%
125	126	0.8%
150	151.67	1.11%
175	177.67	1.52%
185	210.33	13.69%
200	255	27.5%

Table 3.2. Distance test for LEGO NXT Sonar Sensor 1.

Reference Distance	Measured Distance	Relative Deviation
5	7	28.57%
10	12	16.67%
20	21	4.76%
40	40	0%
60	60	0%
80	80	0%
100	100	0%
125	125	0%
150	150	0%
175	175	0%
185	185	0%
200	200	0%

Table 3.3. Distance test for LEGO NXT Sonar Sensor 4.

3.5 LEGO IRSeeker

The HiTechnic NXT IRSeeker V2 is a sensor able to detect an infrared signal and return the direction of the source. The IRSeeker has five infrared receivers placed at 60° intervals. Based on the strength of the signal received at all receivers it calculates the direction of infrared signal's source. The IRSeeker has nine sectors as can be seen in Figure 3.5. It will return one of these sectors depending on the calculation of the direction. The five sensors are placed in sector 1, 3, 5, 7, and 9. The IRSeeker can furthermore return the strength of the IR signal on the five sensors.

The IRSeeker has two modes:

- **Modulated (AC)** mode detects modulated IR, which certain IR transmitters can transmit in. This is a more efficient way of obtaining the direction of the transmitter, as it can filter out other IR light sources, such as sunlight.
- **Un-modulated (DC)** mode is more prone to be affected by sunlight as it receives all IR signals. This is used for transmitters unable to send a modulated signal.

The documentation for the IRSeeker can be found at [4].

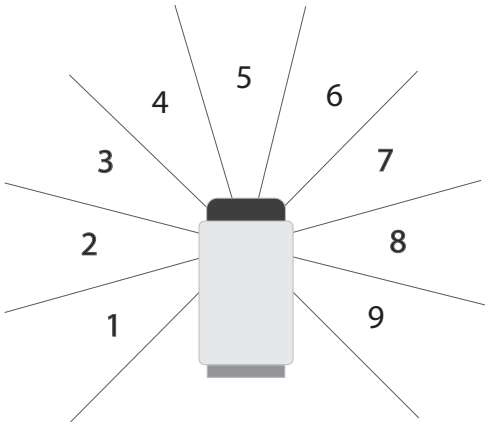

Figure 3.5. LEGO NXT IRSeeker.

3.6 Conclusion

The provided LEGO hardware has been tested in this chapter, and its limitations has been covered. The performance of the LEGO actuators and sensors can be a bottleneck for the future software implementation. The theory behind the two sensors; the sonar and the IRSeeker would suggest that the IRSeeker is the better choice. We will look into creating software for each sensor in Chapter 4.

Detection Methods 4

The description of the different hardware pieces available provides an opportunity to look at detection methods for the project. The detection methods are different approaches for the Stalker to keep track of the Runner. This means describing the main capabilities of the hardware and test its limitations. With the given hardware there are three different approaches available: IR, sonar, and sound. These approaches will be described in detail in this chapter. The end of the chapter will conclude which approach is most suitable with the limitations taken into account.

4.1 IR Detection

IR detection works by placing an infrared signal on the Runner and an IRSeeker on the Stalker as illustrated in Figure 4.1. The IRSeeker works as described in Section 3.5. It uses its five IR sensors to estimate the direction of the IR signal. The estimated direction is based on the strength of the signal measured by the different sensors and is returned as a number between one and nine. Therefore creating software using the IRSeeker is simpler than when using other types of hardware, like the sonar sensors (see Section 4.2).

IR detection does not provide a reliable option to detect the distance between the Runner and the Stalker. Therefore to avoid them driving into each other a sonar sensor must be placed on the Stalker to keep track of the distance to the Runner. In Section 3.4 it was concluded that the sonar sensors are not precise within the whole documented range (5-255). Within the 20 cm to 210 cm range the readings are reliable, however. Thus the sonar sensor is best used at detecting if the cars get too close or too far away from each other.

In conclusion using IR detection is a relatively simple and effective method as it does not require many calculations and thereby the system will be able to respond faster to the stimuli it receives.

4.2 Measuring Direction Using Sonar Distance

Another approach to get the Stalker to follow the Runner, is using the Sonar Distance sensors as illustrated in Figure 4.2. By placing two sensors on the front of the Stalker it can detect a 40 degree view. The two sensors will be placed on either side of the Stalker.

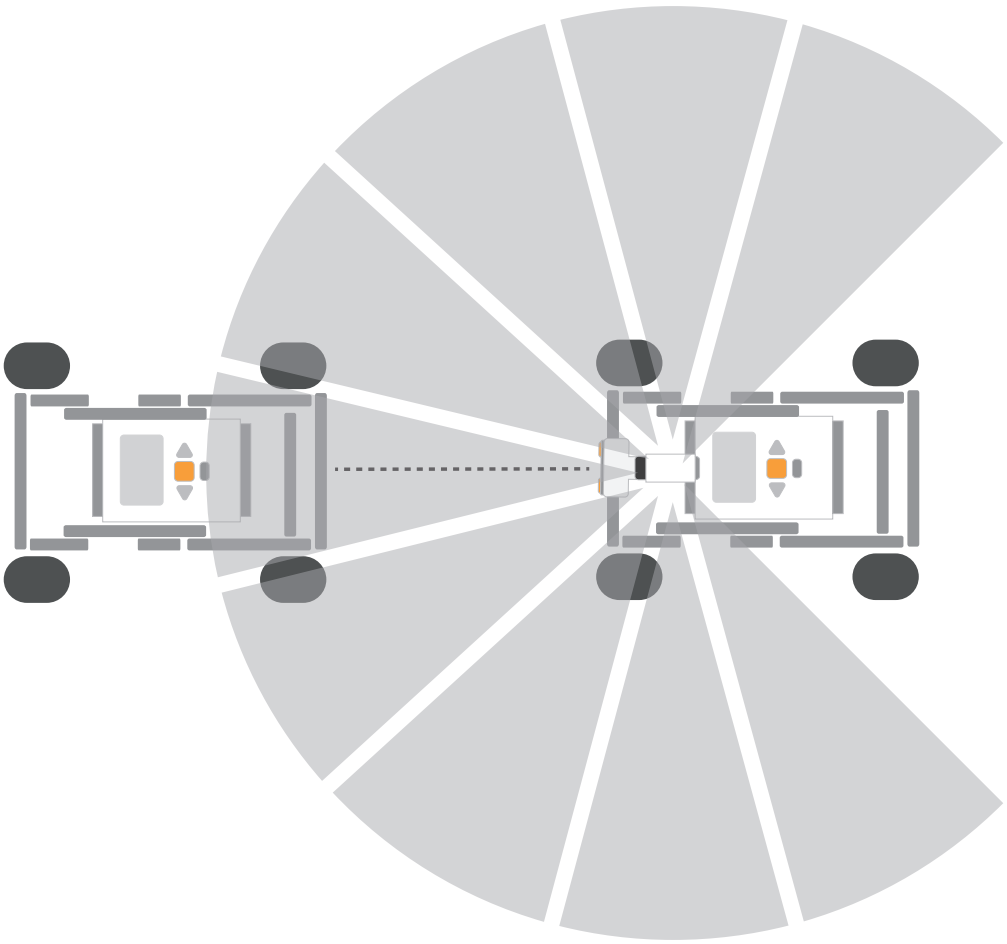

Figure 4.1. A visualization of the IR model.

The left one will point 20 degrees to the left. The right one will point 20 degrees to the right. Using this approach it is able to detect the Runner, given it has a surface the sonar sensors can detect. If the runner turns to the right, the distance sensors on the Stalker will detect the surface on the Runner moving to the right, and adjust its direction accordingly to remain behind the runner. Likewise if the Runner turns to the left. If the Runner simply keeps driving forward it relies on the two side sensors to give approximately the same output, thus telling how far it is from the Runner. This should ensure that the Stalker does not drive into the Runner – accelerating and decelerating the Stalker to keep a certain distance.

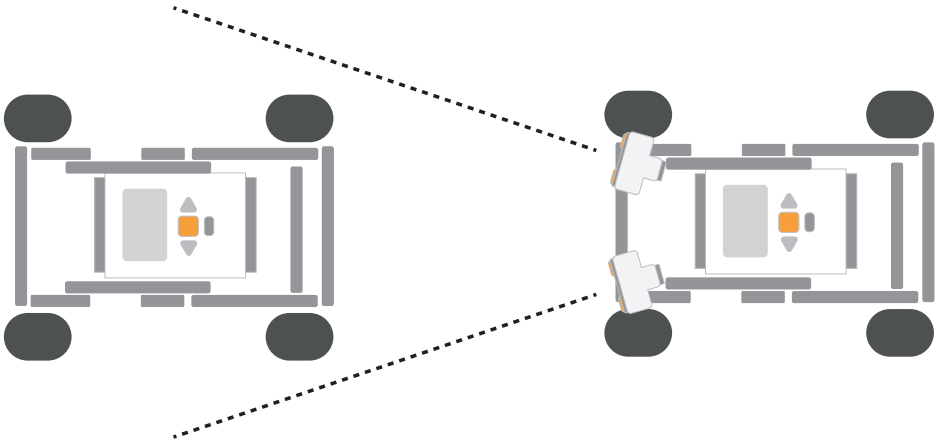

Figure 4.2. A visualization of the Sonar model.

This approach has its advantages and disadvantages. If the sensors detects another surface than the one attached to the Runner, the Stalker will try to *follow* that surface instead. This would result in the Stalker stalking a wall, given that a wall is the new surface it detects. This is a problem as the Stalker would stop following the Runner.

Using this approach would require an initial calibration of the sensors to ensure that they are following the right surface. When the cars start driving, the right and left sensor will measure their distance to the surface and continuously adjust the Stalkers direction to keep these two distances equal. Hereby the Stalker will be able to follow the Runner.

4.3 Bluetooth

Using bluetooth for controlling the cars is an approach that removes the reliance on unreliable sensors and, instead, uses communication between the two cars to make the Stalker follow the Runner. The Runner would use the bluetooth built into the LEGO NXT to send information about its movements to the Stalker, and it would then mimic these movements by making the same function calls. The advantage of bluetooth is that if a reliable and consistent connection can be made between the two cars then they should mimic each others movements with a high precision rate. The disadvantage is that a package might get lost in the transmission or arrive with a delay. If this happens several times during run-time the delays might propagate and put the Stalker several seconds behind the Runner in movements, or it might simply ignore certain commands if a package is lost.

Due to the inconsistency of the bluetooth on the NXT it is not chosen as the solution, as the main focus of the project would shift towards handling protocols instead of creating the system defined in the problem definition.

4.4 Measuring Direction Using Sound

Another idea for measuring where the Runner is going, is to use the LEGO Sound Sensor, illustrated in Figure 4.3.

Figure 4.3. the LEGO NXT Sound Sensor.

The idea is that it is possible to measure the difference in time between when multiple sound sensors detect a sound, and triangulate the position of the sound source from the time difference.

4.4.1 Evaluation of Triangulation

To determine the viability of triangulation it must be examined during realistic conditions. Based on the design of the cars and how closely they must follow each other, it can be estimated how short windows of time should be detectable for this approach to work. Using the speed of sound, and time difference between when the two sensors detect a sound, the approximate position of source of the sound can be estimated.

Figure 4.4 shows a possible setup using a sound source and two sound sensors. Say the difference in the sensors' distance to the source is 20 cm or 0.2 m, which in a real world scenario would be a realistic value. Based on when the sensors detect the sound, they can estimate the position of the sound's source, or which direction to turn to keep the cars behind each other. The calculation is based on the basic physics formula $v \cdot t = s$, where `v` is the velocity, `t` is the time, and `s` is the distance. When applying this formula to the case where v is the speed of sound (343.2 m/s) and `s` is the difference between the sensors' distance to the source. The time `t` is to be calculated. Isolating time in the formula gives $\frac{s}{v} = t$. Using the values the time difference can be calculated: $\frac{(7.4m-7.2m)}{443.2m/s} = 0.58ms$. This value means that the time difference, which can be detected is less then a millisecond using this method on real world values.

However this project is an emulation of a real world scenario, meaning it is the same situation using a smaller scale. A realistic scenario when using the LEGO cars would put the difference in distance at a maximum of two centimeters before an action would need to be taken. Therefore the calculation would be $\frac{(25cm-23cm)}{443.2m/s} = 0.058ms$.

This value is relatively low but not unrealistic in a real-time system, and triangulation is hereby a viable option under optimal conditions. There is, however, a problem with this approach using the given hardware. The sound sensors are not able to detect sound on a specific frequency, but only how loud a sound is. This means that if a sound is to be detected it has to have a specific sound level, or just be in a sound level interval. As there will be background noise, which has an unspecified sound level, basing the detection on triangulation using the given hardware is not a reliable solution, as there are no means to

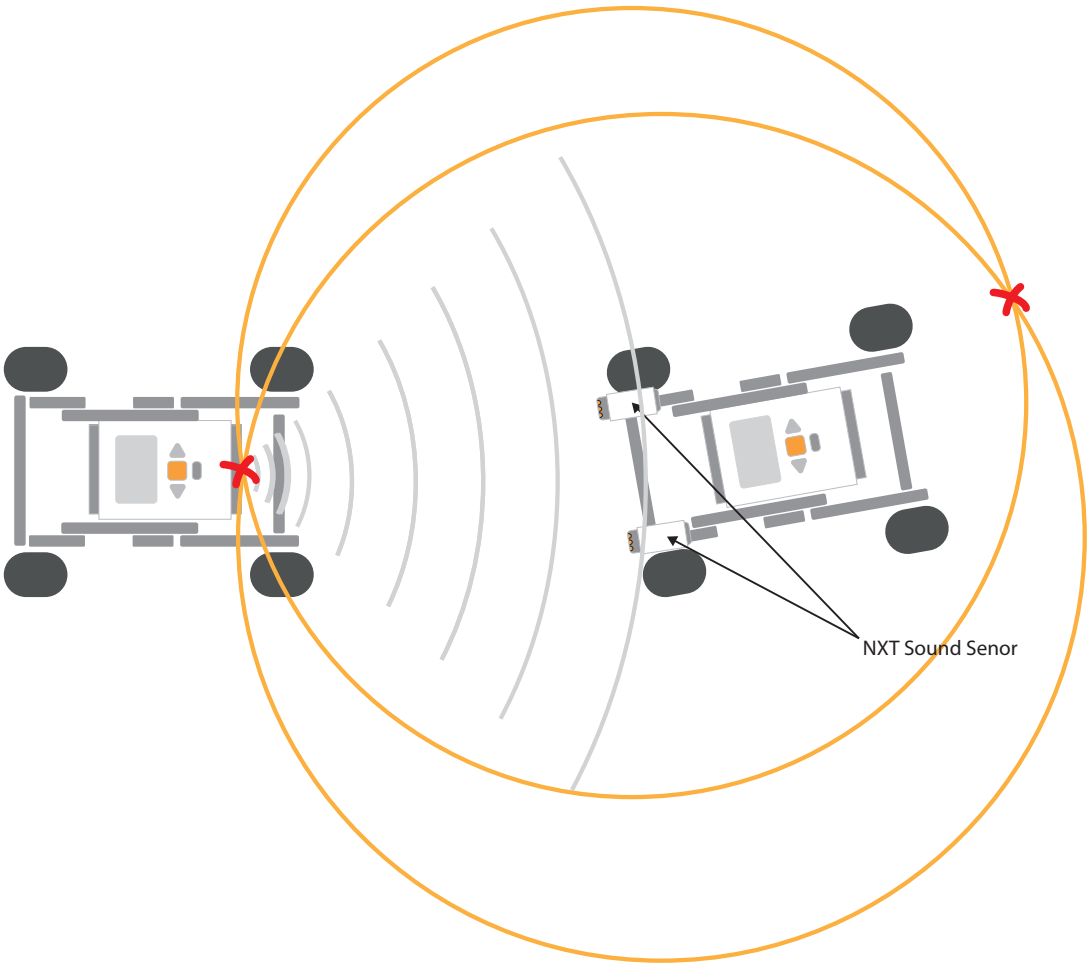

Figure 4.4. Example case of sound triangulation.

guarantee that the detected sound originates from the Runner.

4.5 Conclusion

This chapter discussed some different approaches to detect which direction the Runner moves in. The ideas originate from the hardware mentioned in Chapter 3, and how they could be used to achieve direction detection. The most optimal solution is the IRSeeker as it returns a direction the car can adjust to and it only detects IR signals, where as a sonar sensor e.g. detects all surfaces including walls. The solution with the IRSeeker contains some problems, however. Even though it is the simplest software solution and probably the easiest to base the Stalker upon, nxtOSEK does not have an API for the IRSeeker, thus we will have to implement it ourselves. This is not possible within the given timeframe.

A similar problem exists with bluetooth. While it is an optimal solution since the stalker would simply just mimic the Runners movements, the inconsistency and unreliability of the connection makes it an unviable solution under the given circumstances. As with IR

this would have been a solution during better circumstances.

Therefore as the two most optimal solution are unviable with the given circumstances, we have opted to use the sonar sensors for detection as they provide the best solution which is viable.

We chose not to use the sound sensors, since they as described would detect background noise as well which might cause the Stalker to make adjustments based on these.

Part III

Design

Hardware Construction 5

Both the Runner and the Stalker will be based on the same hardware model. It is based on [5], which is a guide that describes how to build this model. A few additions will be made, listed below.

Figure 5.1. LEGO car based on instructions from [5].

When looking at the basic model of the car in Figure 5, the following changes will be made to the car:

- **Sonar Sensors** – As mentioned in Section 4.5, a total of five sonar distance sensors will be added to the cars. Three will be added to the Runner and two to the Stalker. The three on the Runner will make sure it does not drive into various obstacles, e.g. a wall. Two sensors looking to the sides and one straight ahead. The two sensors on the Stalker is for detecting the surface (see 3rd bullet) on the Runner and follow this. By measuring which sensor has the most distance to the surface, it can be detected whether the Runner is turning and in which direction. If so the Stalker must turn as well.
- **Steering** – The design from [5] specified that the steering motor was connected directly onto the steering mechanism. To enable more precise turning, a gearing of this motor was added. The steering will be geared with 60%, meaning that to turn the wheels 100 degrees the motor has to achieve a count of 166.67. This is achieved

by attaching the motor to a 24 cogwheel gear and the steering to a 40 cogwheel gear. These two cogwheels will be connected to each other. The gearing is found using the fraction $\frac{24}{40} = \frac{3}{5}$. Thus giving a 60% gearing. The ratio between the degrees turned and the motor count can be represented by the following equation: let y be degrees and x be the motor count in $\frac{3}{5}x = y \Leftrightarrow x = y\frac{5}{3}$. Using this formula degrees y is always obtainable as long as the motor count is known.

- **Surface on the runner** – In order for the Stalker to be able to follow the Runner, it must have a surface that the sonar distance sensors can detect and keep detecting. A few bricks will be used to make a holder for an A4-sized cardboard, which will be placed on the back of the Runner.

These changes are sufficient for the cars to get the functionality needed.

Physical Constraints 6

Even though the hardware design is simple and robust, it is necessary to test any possible constraints that might occur, so compensations for these can be implemented in the software.

6.1 Brake Distance Test

The purpose of this test is to measure the brake distance of the LEGO car construction. The brake distance helps describing how the implemented function `motor_set_pwm(port Port, int PWM, bool Brake)` physically affects the car, when parsed a boolean being true.

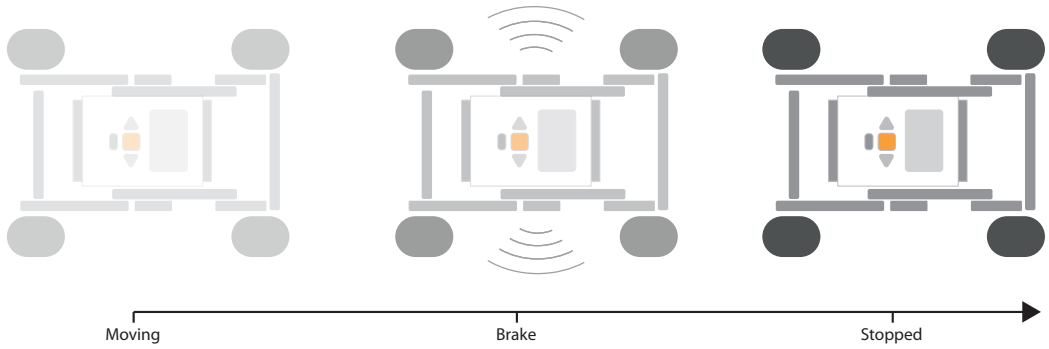

Figure 6.1. The brake distance test setup.

The test setup is illustrated in Figure 6.1. The setup consist of a LEGO NXT car, which runs a task. The task sets the driving motor at a given PWM for two seconds, which causes the car to move forward, and then enables the brake function. Just before the braking is enabled the car plays a sound, to determine when it starts breaking. The source code for the task can be found in Appendix A.1. The car drives, while being filmed, next to a measuring tape, which is placed as a reference distance for the analysis. After the filming phase, the videos are exported and edited in video editing software (Adobe After Effects). The editing consists of stabilizing the video footage in order to reduce the shaking from the handheld camera, and detect the exact point in time where the brake is toggled by

looking at the waveform of the sound.

After the stabilizing phase, the video is then analyzed using the tracking software *Tracker* [14]. Data containing information of coordinates at a given time is then retrieved, and is used further for calculating the brake distance. The results can be seen in Table 6.1.

PWM	Test 1 (cm)	Test 2 (cm)	Test 3 (cm)
65	0,80	1,83	2,15
75	4,14	5,39	3,53
85	3,26	3,63	3,58
100	8,15	8,01	8,24

Table 6.1. The brake distance test results.

Test 1 was captured at a further distance than test 2 and 3. The measuring tape was extended to 100 cm, while during test 2 and 3 it was only extended to 50 cm. The shorter length of the measuring tape made it possible to get the camera closer, and the precision during the tracking increased. Precision increased because having the camera closer gives a higher pixel density on the measuring tape in the video, which in turn makes tracking the car more precise. Excluding the 75 PWM results, a polynomial behavior can be seen in the test results.

The braking times can be found in Appendix A.4, but as they will not be used in the project they will not be elaborated further.

Sources of Error

When performing the test there are several potential sources of error, which can explain the inconsistency in the test results:

- Clicking in the tracking software — The tracking software requires one to click on a certain spot of an item repeatedly as the item moves, so one can see how it moves. The problem arises if the click is slightly inaccurate and does not hit the exact same spot on the car every single time. This source of error will cause the braking length to become either smaller or greater than it is supposed to.
- Blur on the video — As the camera is not high speed it will have some blur. This blur makes the clicking in the tracking software even harder, and thus added to that source of error. This error will not directly affect the braking length measured, but could, as explained, cause the clicking in the tracking software to become less accurate.
- Sampling frequency — As the video has a lower sampling frequency for frames than the sound, the car might start playing between two frames, thus it is impossible to tell exactly where the car starts breaking. This will cause the tracking software to measure an incorrect breaking length.
- Speed of sound — This is more of a theoretical source of error. As the distance between the car and the recording device is less than 0.85 meters. The time it would take the sound to get from the car to the recording device would be less than 2.5 ms, this is not considered a relevant source of error.

To avoid the sources of error, alternative tests could have been performed, had the needed equipment been available. Using an accelerometer a more precise time stamp for when the braking began could be found, with better recording equipment a more precise braking length could be found, or by having the car brake at a specific point and just measure how far it gets before stopping.

However as the needed equipment for these tests were not available the conducted test was chosen.

6.2 Velocity

The purpose of this test is to measure the velocity of the hardware construction, which is described in Section 5. The test was conducted using the same data and setup as in the Brake Distance Test, which is described in more detail in Section 6.1. An initial test showed that the construction was not able to move at the PWM interval $]-63, 63[$ meaning in the interval from -63 through 0 to 63, excluding both 63 and -63. The data is represented in the form of x and y coordinates at time T. The x and y coordinates are converted into vectors, and their length is computed and summed up. The total sum of the vector lengths is then divided by the time interval they represent. Three tests have been done at each of the chosen PWM values.

PWM	Average velocity, V
65	$16,23\frac{cm}{s} \Rightarrow 0,58\frac{km}{h}$
75	$29,60\frac{cm}{s} \Rightarrow 1,06\frac{km}{h}$
85	$44,33\frac{cm}{s} \Rightarrow 1,59\frac{km}{h}$
100	$57,59\frac{cm}{s} \Rightarrow 2,07\frac{km}{h}$

Table 6.2. Measured velocity at given PWM.

The average results based across the three tests can be seen in Table 6.2. For further documentation about the calculations, the Velocity Test Section A.5 can be viewed.

Steering Approach 7

The cars will be steered based on the input they read from the sonar sensors. A number of methods exist that can be used to control the cars. The approaches can be divided into two categories:

One where the Stalker reads some values. Based on these values it estimates the specific action needed to follow the Runner. The other where the Stalker is controlled dynamically. The steering is based on how the input changes, i.e. if the Runner turn the Stalker begins turning, and if the Runner stops turning, so does the Stalker.

7.1 Steering Based on Estimations

Several approaches exists to steer the cars based on estimations and calculations. The information available to do these will be the two distances measured by the sensors, the count of the steering motor and the current PWM of the driving and the steering motor. Based on this information there are two main options for steering the Stalker.

One which is based on estimating the angular acceleration of the Runner's turn, and one based on estimating the angle with which the Runners wheels are turning and emulate it.

7.1.1 Angular Acceleration

The discussed angle is illustrated in Figure 7.1. To use angular acceleration a history of the angles must be kept, as at least two values are needed to calculate it. To calculate the angular acceleration the angular velocity must be known. It is calculated by measuring how many degrees the angle changed over a known period of time using the formula: $\omega = \frac{\theta}{t}$, where ω denotes the angular velocity, θ denotes how many degrees the angle has changed, and t the time. The angular acceleration is then obtained by using the formula: $\frac{\Delta\omega}{t} = \alpha$, where $\Delta\omega$ denotes the change in velocity, t the time and α the angular acceleration. This assumes a constant angular acceleration, so the system would have to be designed to account for this. By knowing the angular acceleration with which the Runner is turning, the Stalker can be programmed to emulate this angular acceleration, and hereby mimic the Runners movements.

Precise and reliable measurements are needed to use this approach, as otherwise the Stalker might emulate the wrong angular acceleration.

7.1.2 Angle Estimation

Angle estimation refers to calculating the angle the Stalker's wheels should be in. The approach is to calculate the angle of the Runner compared to the Stalker and set the wheels of the Stalker in this estimated angel relative to the Runner. The calculations are done by creating a geometrical model of the two cars position as illustrated in Figure 7.1, and based on it obtain the necessary values to compute the desired angle. The calculations would be done using the sine- and cosine-relations.

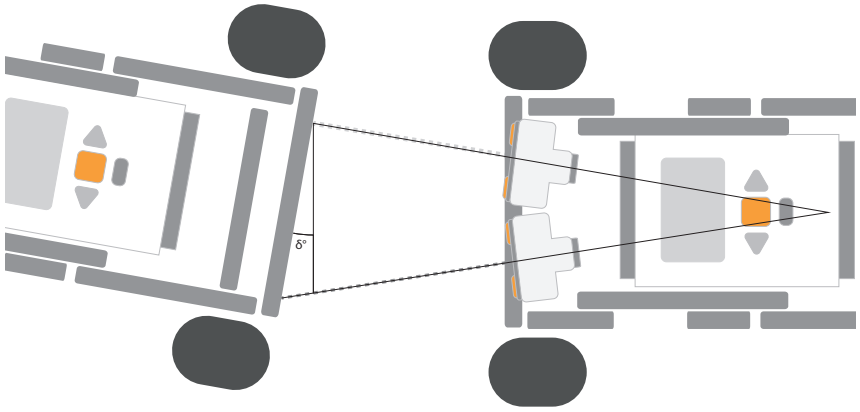

Figure 7.1. Angle estimation illustrated.

Positioning the wheels in the correct position will be handled by a task with a short period, where the position is continuously checked and the motor stopped when the wheels reach the correct position.

The advantage is that an angle can remain consistent through different measurements, i.e. the value pairs $(20, 15)$ and $(26.667, 20)$ will yield the same angle. Therefore even though the values measured are different, the angle is consistent, i.e. the Stalker needs no adjustment.

Angle estimation is a desirable approach as it gives precise data about the cars' current direction compared to each other, regardless of how high or low the raw values are.

7.2 Dynamic Steering

When using dynamic steering the Stalker will react to the raw values. If the sensors indicate that the Runner is turning then the Stalker should begin to turn as well. Furthermore, when the raw values indicate that the Runnerstops turning the Stalker should as well.

Physical restraints are needed to use dynamic steering effectively. The turning speed must be limited as otherwise it might cause the Stalker to turn too much compared to the Runner. If these physical restraints are not in place the Stalker might turn to much before detecting it should stop turning, causing a zigzagging motion as it continuously try to get behind the Runner. Thus the main disadvantage of the approach is that it must be restricted to work properly.

7.3 Choice of Steering

The chosen steering model for the implementation is a modified dynamic steering model. The steering model will be based on dynamic steering combined with other aspects.

Two histories will be kept, one containing the values read, and one containing the angle discussed in Section 7.1.2. Based on how the angle has changed over time the Stalker will then begin turning if the angle history indicates a turn. Likewise it will lock the steering wheels, if the angle history indicates no significant changes has occurred. If the angle history indicates that the Runner has stopped turning, then the Stalker should stop turning as well.

An issue that must be dealt with in this approach is the unreliability of the sensors. Due to these, the angle calculated might be incorrect, and therefore a tolerance should be implemented. This tolerance should state that if the angle is smaller than a certain value, no steering adjustments has to be performed.

Software Platform 8

The following real-time system operating systems will be examined: leJOS NXJ and nxtOSEK. The examination will focus on the real-time system aspects needed to implement the desired system. The two operating systems will be evaluated according to how well they support **tasks**, **timers**, **error handling**, and **scheduling policy**.

8.1 leJOS NXJ

leJOS is a Java programming environment for the LEGO MINDSTORMS NXT [6]. It provides an API that contains the functionality needed to program the different Mindstorms and HiTechnic sensors.

8.1.1 Tasks

Tasks are represented in leJOS by objects which inherit from the `java.lang.Thread` class. The `java.lang.Thread` class has been expanded to include the method `setPriority()` that makes it possible to give tasks a dynamic priority because it can be modified during run-time.

8.1.2 Timers

Timers are needed in a real-time system to control the release of periodic tasks. leJOS has its own timer class and an associated timer-listener class which listens for when timers are triggered.

8.1.3 Error Handling

leJOS has two ways of handling errors and debugging them: Exceptions and Data Aborts. Most of the standard Java languages exception classes are supported by leJOS. Users are also able to create their own classes to handle errors using exceptions. Data aborts only occur if the operating system crashes.

8.1.4 Scheduling Policy

The scheduler will always run the highest priority thread in preference to any others. If more than one thread of that priority exists, the scheduler will time-slice them. In order

for the lower-priority threads to run, a higher-priority thread must cease to be runnable. Therefore it must either exit, sleep, or wait for a monitor. Yielding is not sufficient. This implies that leJOS uses fixed-priority scheduling with preemption as higher-priority tasks might wait for a monitor to indicate a resource held, by a lower-priority task, has been released, i.e. the higher-priority task is in a preempted state.

8.2 nxtOSEK

nxtOSEK is a C/Assembly source code programming environment for the LEGO MINDSTORM NXT. It consists of the device drivers of leJOS NXJ, TOPPERS/ATK (Automotive Kernel, formerly known as TOPPERS/OSEK) and TOPPERS/JSP Real-Time Operation System that includes ARM7 specific port parting, and finally some custom code to make it all work together.

8.2.1 Tasks & Timers

In nxtOSEK, all real-time aspects, such as tasks, alarm and timer definition, settings, and configurations, are defined and configured in the provided OIL (OSEK Intermediate Language) file, which has the .oil file extension. Regular code and the actual content of the various tasks is defined in a regular C file. The file contains regular C-code with a few needed modifications, such as the `TASK` keyword used to define tasks.

nxtOSEK includes a large library of functions which can be used to interact with the different sensors and motors. These are simply implemented as functions that can be called from the tasks.

8.2.2 Error Handling

nxtOSEK has no built-in features for handling errors. It is, however, possible to implement a watchdog. A watchdog measures a given tasks execution time.

8.2.3 Scheduling Policy

nxtOSEK uses a Fixed Priority Scheduling (FPS) scheme, which both supports periodic, aperiodic, and sporadic tasks, using OSEK Alarm Counters and events to release aperiodic tasks. [10]

Due to the implementation of the OSEK Alarm Counter, tasks cannot have a shorter period than 1 ms. To avoid priority inversion, nxtOSEK uses the Original Ceiling Priority Protocol (OCPP). [9] [11, p. 31]

8.2.4 Platform Selection

We chose nxtOSEK for the following reasons:

- Deallocation of memory is easier to control in a language that is not designed to use a garbage collector.
- It is our impression that nxtOSEK has better documentation, e.g. there is no explicit description of the scheduling policy for leJOS.

Any references to API's and built-in functions will therefore be in respect to the API's and built-in functions that nxtOSEK and C offer.

As a side effect of choosing nxtOSEK, the FPS scheme will be used.

Software Design 9

A major part of the design involves dividing the functionality into different tasks. It should also be considered what functionality might be reused across the tasks and place this functionality in functions. Aside from this it must also be decided how to implement the functionality since the solution to how the car must turn is non-trivial. The development of this will be explained in this section and the implementation in Part IV.

> " The simplest solution is often the most robust. "
>
> — René Rydhof Hansen, AAU

With the above quote in mind, it is advised when designing a real-time system to divide the program into the smallest tasks possible while remaining reasonable. Reasonable is subjective, and should be considered within the context of the given program. Smaller tasks make the scheduling of the system simpler. Tasks will have lower costs and thus less preemption is needed, because they will generally finish before higher-priority tasks need computation time. This was explained in further detail in Section 2.3.

The main functionality of the system is:

- The Runner must not collide with walls or other obstacles.
- The Stalker must keep a safe distance to the Runner.
- The Stalker must follow the Runner.

These are the main functionalities of the system. However, there are subparts to these functionalities that will be elaborated on in the description of the individual cars.

9.1 The Stalker

The Stalker will have the tasks described below.

- **Initialize** – Initializes the car by calibrating the steering and gathering some initial information from the sonar sensors. This task is only run once and is the first task to run.

- **RightMeasure** – Polls the right sensor for data and adds the data to a history of readings for the right sensor.
- **LeftMeasure** – Polls the left sensor for data and adds the data to a history of readings for the left sensor.
- **Steering** – Decides which direction the Stalker should travel, based on the history of readings.
- **Wheels** – Makes sure that the wheels are turned as much as they should, based on the calculations made by the **Steering** task.
- **Distance** – Makes sure that the Stalker keeps an appropriate distance to the Runner by accelerating or decelerating as necessary.
- **Watchdog** – Measures the running time of the other tasks and checks if they exceed their *WCET* (Worst Case Execution Time).

The purpose of the **Initialize** task is to reduce human errors in the system, so the car does not have to be positioned exactly the same each use. The calibration makes sure that the car takes the environment into consideration. Additionally, the **Initialize** task fills the history of measurements with initial values.

As mentioned in Section 3.2 the motor has an internal counter that is incremented by one for every degree it turns in one direction and decremented by one for every degree it turns in the other direction. Unless the motors are given the same direction for the count `0` they will not be driving in the same direction when having the same count. As well as having the same perception of what driving in a straight line means, the Stalker must know the maximum amount it can turn in both directions. This is achieved by having the car turn the maximum possible amount to the left and the right during the calibration.

The other tasks are self-explanatory except for the **Watchdog**. The **Watchdog** watches over the other tasks and checking whether they meet their respective deadlines or not. If they do not, it will execute an alarm that warns the driver that the auto-driving has failed and he should take over and/or reset the system.

9.1.1 The Distance Task

Keeping a fixed distance to the Runner makes it simpler to make the appropriate turns. In case the Stalker is too far away it might lose vision of the Runner. If the Stalker comes too close to the Runner it might crash into it. The focus of the **Distance** task is keeping the appropriate distance to the Runner.

This so-called *appropriate distance* is a distance determined by experimentation and the brake distance test in Chapter 6.

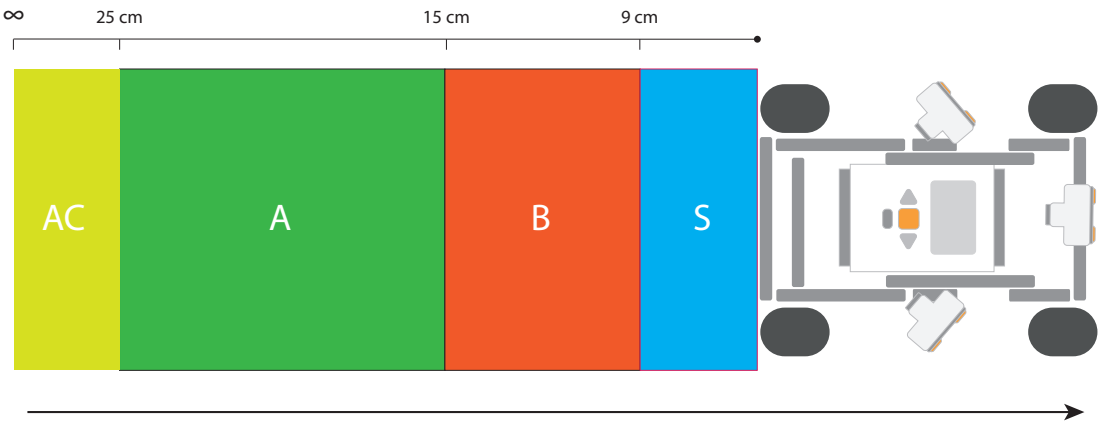

Figure 9.1. The different zones used to keep the appropriate distance. The arrow shows the driving direction.

Figure 9.1 shows the different zones, and their sizes. The zone marked as S is *the stop zone*. Being in this zone is dangerous for the Stalker and it should therefore stop the driving engine. The distance 9 cm is chosen from the brake distance test in Section 6.

The next zone, marked B, is *the brake zone*, which makes the Stalker decelerate. All following distances are assessed by eye estimation, based on the relationship between the size of the A4-sized cardboard and the degree on which the sensors are pointing away from each other on the Stalker, such that the sensors can always see the cardboard.

The next zone, marked A is *the appropriate zone*. The distance from the sensor to the surface has been estimated by eye. In this zone the two sensors on the Stalker will be able to detect the Runner, as described in Section 7.

The final zone, marked AC, is *the acceleration zone*. This is an undesired zone and the Stalker must get out of it as fast as possible by accelerating.

9.2 The Runner

The Runner will have the following tasks:

- **Initialize** – Initializes the car, calibrating the steering and adding sonar sensor measurements to the history of measurements.
- **Steering** – Polls data from all three sonar sensors and bases its decision to turn upon these measurements.
- **Wheels** – Ensures that the wheels are set at a certain degree according to the data from the **Steering** task.
- **Watchdog** – Measures the running time of the other tasks and checks if they exceed their WCET.

The **Initialize** task has the same purpose as the Stalker's, described in Section 9.1. The turning task has been included to avoid using busy-wait in the **Steering** task, as busy-wait

could potentially skew the computation time of the task each time it is run.

As with the Stalker, the actual implementation of the Runner can be seen in Part IV.

9.3 Run-time Estimations

There is an **Initialize** task in both the Runner and the Stalker. Its purpose was described in Section 9.1. The **Initialize** task is special because it is the only task that is run once, and will have a period of 0 ms.

The estimations of the remaining tasks' periods are based on assumptions of how they are expected to run. This will be tested and reasoned for later in the report.

9.3.1 Runner

The **Steering** task feeds information to the **Wheels** task. Constant surveillance of the system is needed in order to have the wheels in the desired position. In order to accomplish this behavior, we assume that we need to perform the computation more often than the **Steering** task. Therefore the **Wheels** task must be run more often. Monitoring the executions of all the tasks is done by the **Watchdog** task. It is required to run often in order to monitor the tasks.

Initially, it is deemed to be necessary to run the **Steering** task every 200 ms. The **Wheels** task will run four times between every execution of the **Steering** task. Its initial period is set to 50 ms. The **Watchdog** task will run every 50 ms to ensure it properly detects WCET overruns every time the tasks are run.

The initial task set for the Runner is as seen in Table 9.1.

Task	Period (ms)
Initialize	0!
Steering	200
Wheels	50
Watchdog	50

Table 9.1. Estimated periods for the Runner.

9.3.2 Stalker

The **RightMeasure** and **LeftMeasure** tasks receive input from the sonar sensors, and filters the data, deleting invalid measurements such as *spikes*. They also provide data for the **Steering** task. Therefore they must run more often than the **Steering** task. The **Steering** task and the **Wheels** task work as in the Runner and are given the same relation between their periods. The **Distance** task does not need to run often due to the distance it travels in a second. Therefore it will have a longer period than, e.g. **Wheels**, which needs a short period as it is constantly correcting controlling the wheels of the construction. The **Watchdog** task monitors the executions of all the tasks, and is required to run often in order to check the status of the tasks. Therefore it will have the shortest period.

The **RightMeasure** and **LeftMeasure** tasks will have periods of 100 ms, i.e. half the period of the **Steering** task. The **Watchdog** task will initially have a period of 50 ms to ensure it monitors all tasks execution time.

Task	Period (ms)
Initialize	0!
RightMeasure	100
LeftMeasure	100
Steering	200
Wheels	50
Distance	200
Watchdog	50

Table 9.2. Estimated periods for the Stalker.

The periods for all the tasks in both the Runner and Stalker are, at this point, estimations of how often the tasks should run in order for all tasks to run often enough to do their respective jobs. There are certain dependencies which are also taken into account and reflected in the periods of the tasks, e.g. the Runner's **Wheels** task depends on the result from the **Steering** task, while the **Steering** task depends on input from sensors received via the **RightMeasure** and the **LeftMeasure** tasks.

9.4 Watchdog

The **Watchdog** task is used to handle situations where a task's execution takes longer than its calculated WCET. Both the Runner and the Stalker will have a watchdog. It should run often enough to detect if the task with the smallest period exceeds its WCET, but not so often that it will influence the behavior of the system while all tasks run within their WCET. Note that the watchdog itself will not be monitored.

In the estimations shown in Table 9.1 and 9.2 the shortest period for both the Runner and the Stalker is 50 ms. The **Watchdog** task should run every 50 ms as this is the shortest period in both the Runner and the Stalker.

Part IV

Implementation

Software Construction 10

This part describes how the concepts and ideas from Part III are implemented. Due to the choice to use nxtOSEK, functionality will be included in the solution which is nxtOSEK-specific and not project-specific. Because of this it was not covered in Part III.

10.1 The Runner

In the following sections, the code for the Runner will be explained.

Final Task Set

The periods for the Runner's tasks were shortened significantly compared to the estimated periods mentioned in Section 9.3. During development, it became clear that the periods were too long because the Runner was reacting too slowly and thus did not behave as intended. The **Wheels** task's period was shortened to 5 ms, and accordingly The **Watchdog** task's period was shortened to account for this. The deadlines for the tasks were set to be their period.

Task	Period, T	Deadline, D	Priority, P
Watchdog	5	5	3
Wheels	5	5	2
Steering	30	30	1

Table 10.1. Task set for Runner. 1 is the lowest priority.

Global Variables

In the code for the Runner there are some global variables, which are used across several tasks. These variables are:

- `maxturn` contains a value set by the **Initialize** task that is the maximum count that the steering motor can reach.
- `midturn` is calculated by dividing `maxturn` by two. It describes the count at which the wheels are pointing straight forward.
- `drivespeed` is the speed that the Runner is driving at.

- `wantedAngle` is the value the Runner must turn. This value is set in the **Steering** task and is used in the **Wheels** task.
- `turner` describes how much the Runner will turn. Depending on the `wantedAngle`, it is either set positive or negative to turn left or right.

Furthermore, the following global variables are present: `w_starttime`, `w_running`, `w_wcrt`, `s_starttime`, `s_running`, and `s_wcrt`, which will be elaborated further in the description of the **Watchdog** task.

The code also contains a number of defines, made to make it more readable:

- `MOTOR_STEERING` is used instead of `NXT_PORT_A`, which is the motor used for steering the car.
- `MOTOR_DRIVE` is used instead of `NXT_PORT_B`, which is the motor used to drive the car either forwards or backwards.
- `SONAR_LEFT`, `SONAR_FRONT`, and `SONAR_RIGHT` corresponds to `NXT_PORT_S1`, `NXT_PORT_S2`, and `NXT_PORT_S3`, respectively. The names describe the placement of the sonar sensors on the Runner.

The Initialize Task

Whenever the Runner is started the **Initialize** task runs once. This task is only run at the beginning of the program.

The purpose of the **Initialize** task, which can be seen in Code snippet 10.1, is to measure the wheels' maximum possible turning range, calculate the center, and align the wheels to a perfect straight forward position. To ensure the Runner drives straight the **Initialize** task is required so the car is able to reset the wheels to a straight-forward position.

Since this task is only run once directly after starting the program, it is deemed acceptable that it contains a busy-wait, as it will not disrupt the scheduling of the system with its long computation time. The busy-wait is used to turn the wheels straight forward based on the previous measurements.

The last function call of the **Initialize** task is to set the car to start driving (line 29), after this the other tasks are allowed to run.

The Watchdog Task

Code snippet 10.2 shows the global variables used by the **Watchdog** and the tasks being monitored. The first three variables are used by the **Wheels** task, and the last three by the **Steering** task.

The idea behind the implementation is whenever a task is started the task sets its `_running` variable to `1` and updates its `_starttime` variable with the current system time in milliseconds. Each task has its own set of variables with the `w` prefix denoting variables for the **Wheels** task, and `s` denoting the **Steering** task. This is done using the OSEK API function `U32 systick_get_ms(void)` [19], which returns the current amount of milliseconds since the NXT was booted, and not since the OSEK application was

```
nxt_motor_set_speed(MOTOR_STEERING, -100, 0);
systick_wait_ms(1000);
nxt_motor_set_speed(MOTOR_STEERING, 0, 1);

// left
nxt_motor_set_count(MOTOR_STEERING, 0);

nxt_motor_set_speed(MOTOR_STEERING, 100, 0);
systick_wait_ms(1000);
nxt_motor_set_speed(MOTOR_STEERING, 0, 1);
systick_wait_ms(200);

// measure
maxturn = nxt_motor_get_count(MOTOR_STEERING); // global
midturn = maxturn / 2; // global

// steer forward
nxt_motor_set_count(MOTOR_STEERING, 0);
nxt_motor_set_speed(MOTOR_STEERING, -60, 0);

while (nxt_motor_get_count(MOTOR_STEERING) > -midturn)
{
    // busy wait
}

nxt_motor_set_speed(MOTOR_STEERING, 0, 1);
nxt_motor_set_count(MOTOR_STEERING, 0);

nxt_motor_set_speed(MOTOR_DRIVE, drivespeed, 0);
```

Code snippet 10.1. The Runner's Initialize task.

```
// watchdog timers
int w_starttime = 0;
int w_running = 0;
int w_wcrt = 5; // taken from the period

int s_starttime = 0;
int s_running = 0;
int s_wcrt = 30; // taken from the period
```

Code snippet 10.2. Global variables used by the Watchdog.

executed. When the task is done executing, it sets the `x_running` variable to `0` before calling `TerminateTask()`. An example is shown in Code snippet 10.3.

Code snippet 10.4 shows the implementation of the **Watchdog** task in the Runner. By calling `systick_get_ms()` it is calculated how long each task has been running. The `+1` to the `now` variable is needed due to the implementation of the `U32 systick_get_ms(void)` function. The integer, which the function returns, is only increased on each 1000 Hz count. This means that if the function is called 200 cycles after the last increase, then the time which has elapsed since the last increase will not be taken into account. The `+1` will take this overhead into account. The implementation of the `systick_get_ms()` function can be seen in Code snippet C.1.

Whenever a WCET overrun is detected, a tone is played from the brick. This would, if the system was a personal car, alert the driver that he/she should disable the automatic

```
TASK(Wheels)
{
    // tell the watchdog that were are starting now
    w_starttime = systick_get_ms();
    w_running = 1;

    ...

    // tell the watchdog that we're done
    w_running = 0;
    TerminateTask();
}
```

Code snippet 10.3. Example of how monitored tasks update the Watchdog's global variables.

```
TASK(Watchdog)
{
    int now;
    int t;

    if (w_running == 1)
    {
        now = systick_get_ms() + 1;

        t = (now - w_starttime);

        if (t > w_wcrt)
        {
            ecrobot_sound_tone(600,500,100);
        }
    }

    if (s_running == 1)
    {
        now = systick_get_ms() + 1;

        t = (now - s_starttime);

        if (t > s_wcrt)
        {
            ecrobot_sound_tone(600,500,100);
        }
    }
    TerminateTask();
}
```

Code snippet 10.4. The Watchdog implemented in the Runner.

steering and steer the vehicle by himself/herself. The tone is played because this system would be considered a hard real-time system, and when a deadline is missed in a hard real-time system, it means the system has failed, and therefore the driver should be alerted immediately to avoid crashing.

The **Watchdog** task will be tested by using the code in Code snippet 10.5 to increase the cost of the monitored tasks.

```
int j;
int p;
int ii;
int i;
for (i = 0; i < 1000000000; i++)
{
    j = 523432;
    p = 5551132;

    j = (int)((float)p/(float)j);
}
ii = j;
```

Code snippet 10.5. Code used to increase the computation time of tasks.

The Wheels Task

The **Wheels** task controls the actual turning of the front wheels to make the car turn. The **Steering** task sets the `wantedAngle` variable which contains information about the desired angle for the wheels. This is used with the `measured` variable, which contains the current angle of the wheels, to decide whether to turn the wheels or not.

```
int measured = nxt_motor_get_count(MOTOR_STEERING);
int tolerance = 1;
int speed = 63;

if (wantedAngle == 0 &&  measured <=tolerance && measured >= (-tolerance))
    {
    nxt_motor_set_speed(MOTOR_STEERING, 0, 1);
}
else if (wantedAngle == 0 && measured < (-tolerance)) {
    nxt_motor_set_speed(MOTOR_STEERING, speed, 0);
}
else if(wantedAngle == 0 && measured > tolerance) {
    nxt_motor_set_speed(MOTOR_STEERING, -speed, 0);
}
else if(wantedAngle < measured) {
    nxt_motor_set_speed(MOTOR_STEERING, -speed, 0);
}
else if(wantedAngle > measured) {
    nxt_motor_set_speed(MOTOR_STEERING, speed, 0);
}
```

Code snippet 10.6. The Wheels task in the Runner.

The source code for the **Wheels** task can be seen in Code snippet 10.6. There are five checks that decide whether to start turning or not. One could argue for a busy-wait approach to be taken. However, since this would make the computation time for the task dynamic instead static, this would also make it more complex to verify whether the system is schedulable or not.

The **Wheels** task will have a short period. Every time it runs, it checks the current measurement compared to where it wants to go. nxtOSEK does not have a command for making the motors move to a specific position. Therefore it is only possible to start turning and when the task runs again it checks if the measurement and wanted angle match. If this is the case it stops turning.

It has been decided that the motor's PWM is 63 when turning. This value was decided based on tests, that showed this was the minimum value it could receive while it was still able to turn. Any less, and the motor does not have enough power to turn. Any higher, and the motor turns too fast for the system to drive in an acceptable manner.

The Steering Task

This task has to make sure the car does not run into anything in its way. The distances each sensor measures are saved into the `left`, `right`, and `front` variables.

```
1  TASK(Steering)
2  {
3      // tell the watchdog that we are starting now
4      s_starttime = systick_get_ms();
5      s_running = 1;
6
7      int obstacle = 0;
8      int currentcount = nxt_motor_get_count(MOTOR_STEERING);
9      int left = ecrobot_get_sonar_sensor(SONAR_LEFT);
10     int front = ecrobot_get_sonar_sensor(SONAR_FRONT);
11     int right = ecrobot_get_sonar_sensor(SONAR_RIGHT);
12
13     // obsticle in front of us, turn
14     if ((front < 90) && (front != 0)) {
15         obstacle = 1;
16
17         if (right > left) {
18             // turn right
19             ...
20         }
21         else {
22             // turn left
23             ...
24         }
25     }
26     else if (((left != 0) && (left < 50)) ||
27              ((right != 0) && (right < 50))) {
28
29         if (left < right) {
30             // turn right
31             ...
32         }
33         else {
34             // turn left
35             ...
36         }
37     }
38     // as long as we are not in the middle of avoiding a colission
39     else if (obstacle == 0) {
40         wantedAngle = 0;
41     }
42
43     s_running = 0;
44     TerminateTask();
45 }
```

Code snippet 10.7. The Steering task in the Runner.

As seen in Code Snippet 10.7, the task starts by checking if the Runner observes an

obstacle in front of it. It checks for obstacles up to 90 cm away from the Runner using the front sensor. If it measures something within 90 cm, it checks which side has more space – left or right. Then it turns the Runner in the direction of the sensor that measures the biggest distance.

If the front sensor measures more than 90 cm, the left and right sensor will measure their distances. If either measures less than 50 cm, the Runner turns in the direction of the side with more space.

10.2 The Stalker

Now the code for the Stalker will be explained.

Final Task Set

The estimated periods for the tasks were revised during the implementation because the initial conception in Section 9.3 of how fast the cars should respond was too slow. Therefore all tasks were given shorter periods. The **Wheels** task was given a significantly shorter period of 5 ms, since it needs to be this short in order to position the wheels exactly. Aside from that the periods were shortened to ensure the system conformed to the wanted behavior. The tasks' deadlines were set to be their periods.

Task	Period, T	Deadline, D	Priority, P
Watchdog	5	5	6
Wheels	5	5	5
RightMeasure	50	50	4
LeftMeasure	50	50	3
Steering	100	100	2
Distance	100	100	1

Table 10.2. Task set for the Stalker. 1 is the lowest priority.

General Functions and Global Variables

The Stalker's code contains some general functions as well as global variables that are used in tasks as a way to communicate across tasks.

The general functions are: `filterLeftHist`, `filterRightHist`, `degree`, `radianToDegree`, `ourRound`, and `wd_watch`.

`filterLeftHist` and `filterRightHist` filter the recent measurement history from the left and right sensor, respectively. This is done primarily to get rid of the occasional 255, when it is not consistent.

`degree` is the function that fills up the degree history array. These calculations are based on the formula given in Equation 10.1 (using radians). The function is an implementation

of the angle estimation as explained in Angle Estimation in Section 7.

$$\begin{aligned} a &= \sqrt{b^2 + c^2 - 2bc\cos(0.61)} \\ B &= \frac{\arccos(a^2+c^2-b^2)}{2ac} \end{aligned} \tag{10.1}$$

`radianToDegree` is used by `degree` to convert the radian number into a degree, as this is the most useful of the two with the given hardware.

`ourRound` is a function that rounds the number to the nearest integer.

`wd_watch` is a function used repeatedly in the **Watchdog** task and will be explained there.

Below is a list of global variables and a brief explanation of their purpose. A deeper explanation will be given in the following sections.

- `maxturn` is set by the initialize task and holds the maximum count the wheels can turn.
- `midturn` is calculated from `maxturn` and denotes the count which makes the wheels straight.
- `turn` is set in the **Steering** task and dictates how the **Wheels** task works.
- `currspeed` is used by the **Distance** task and contains the speed the Stalker should have.
- `maxspeed` is a global variable that limits the Stalker to a maximum speed, so it cannot go faster than this. This variable is used in both the **Distance** task and the **Initialize** task.
- `lockspeed` is a variable used in the **Distance** task for keeping track of states.
- `leftHist` is an array that contains the recent history of the left sonar sensor. `0`-index in the array holds the newest value.
- `rightHist` is an array that contains the recent history of the right sonar sensor. `0`-index in the array holds the newest value.
- `filteredLeftHist` is an array that consists of the filtered values of `leftHist`. `0`-index in the array holds the newest value.
- `filteredRightHist` is an array that consists of the filtered values of `rightHist`. `0`-index in the array holds the newest value.
- `degreeHist` is an array that contains the most recent history of calculated degrees.

Furthermore, the Stalker has some C preprocessor defines expansion definitions to ease writing. These are:

- `MOTOR_STEERING`, as the name suggests, is the port that the motor used for steering is plugged into.
- `MOTOR_DRIVE` is the port which the motor used for driving is plugged into.
- `SONAR_LEFT` is the sonar sensor measuring on the left side.
- `SONAR_RIGHT` is the sonar sensor measuring on the right side.
- `PI` contains the first 9 digits of pi and is used for the degree calculations.

The Watchdog Task

The watchdog implemented in the Stalker is equal in semantics to the one implemented in the Runner, described in The Watchdog Task in Section 10.1. The only difference is that some of the code of the watchdog has been moved into a function, seen in Code snippet 10.8, in order to avoid duplicate code. The watchdog task can be seen in Code snippet 10.9.

```
1  void wd_watch(int running, int starttime, int wcrt)
2  {
3      int now;
4      int t;
5
6      if (running == 1)
7      {
8          now = systick_get_ms() + 1;
9
10         t = (now - starttime);
11
12         if (t > wcrt)
13         {
14             ecrobot_sound_tone(600,500,100);
15         }
16     }
17 }
```

Code snippet 10.8. Code from the Watchdog task, moved to a function in order to avoid repetition.

```
1  TASK(Watchdog)
2  {
3      wd_watch(r_running, r_starttime, r_wcrt);
4      wd_watch(l_running, l_starttime, l_wcrt);
5      wd_watch(s_running, s_starttime, s_wcrt);
6      wd_watch(d_running, d_starttime, d_wcrt);
7      wd_watch(w_running, w_starttime, w_wcrt);
8
9      TerminateTask();
10 }
```

Code snippet 10.9. The Watchdog task in the Stalker.

The Wheels Task

As specified in Section 9, steering the Stalker will be handled using two tasks: The **Steering** task determines whether the Stalker should turn or not and the **Wheels** task controls the position of the wheels. A global variable named `turn` exists, which is set in the **Steering** task. Based on the value of `turn` the **Wheels** task adjusts the position of the wheels. If `turn` is set to 0 the Stalker will turn straight ahead. If it is set to 1 it means a right turn must be performed, and a value of 2 indicates a left turn.

The first case on line 7 to 17 in Code snippet 10.10 handles the case where the Stalker should drive straight ahead. A tolerance of `10` degrees is used on line 8, 11 and 14. A tolerance is used because having wheels at the count `0` is unlikely to happen, due to the rapid movement of the wheels (`63` PWM). If a tolerance is not used then the driving will

be less accurate since the **Wheels** task will make the wheels continuously turn from left to right, never ending with the count `0`, i.e. straight ahead.

The second case on line 18 to 31 takes care of the situation where the Stalker should turn right. The `if`-statement on line 19 to 21 takes care of the situation where the Stalker has been turning right for a while and is set to turn a bit less to the right. In this situation the wheels should not turn at all, since the more the wheels are turned, the more wobbling behavior will emerge.

This could be a problem if the change in degree is drastically big, since it would prevent the Stalker from turning when it should. This will not be the case since the **LeftMeasure** and **RightMeasure** tasks are released often, and the difference will therefore be sufficiently low.

One could imagine that this code would cause a problem if a continuous change from a sharp right degree to a smaller degree will cause the wheels to never turn at all. This will not be the case, since the sonar sensors are unreliable and returns changing numbers that might jump a bit both in the positive and the negative direction. Therefore a continuous change in degree from positive to less positive over a large period of time will not occur.

The `else if`-statement on line 23 to 25 takes care of the situation where the change in degrees is less than three. Since this number is so small, it is hard to achieve a change in the position of the wheels which is within a maximum of three degrees.

The `else if`-statement on line 26 to 28 takes care of the situation where the Stalker has measured that it should turn less to the right It is assumed that the Stalker is on its way to be fully aligned with the direction of the Runner. Thus it is assumed that it is safe to drive straight ahead, by setting the `turn` variable to `0`.

The `else`-statement on line 29 to 31 takes care of the situation where none of the previous checks have been evaluated to true. Therefore the Stalker should simply turn more to the right.

The above explanation of the code applies to case 2 on line 33. However, as this is identical in behavior except that it is for the left turn, we will not elaborate further, and it has been omitted from the code snippet.

The Measure Tasks

In Section 9.1 the tasks for the Stalker were described, in this section the two measuring tasks – more specifically the **RightMeasure** task and the **LeftMeasure** task, will be looked at in further detail. Since the two tasks, in principal, are similar only the **RightMeasure** task will be described in this section. The only difference between the two tasks is that the **LeftMeasure** task measures the left, thus simply replace “right” with “left” in Code snippet 10.11.

It was decided to have two tasks for measuring the distance on left and right to reduce interference.

The code in Code snippet 10.11 receives some input from the right sonar sensor, which is

```
TASK(Wheels) {
    int measured = nxt_motor_get_count(MOTOR_STEERING);
    int tolerance = 10;
    int speed = 65;

    switch(turn){
        case 0: // go straight
            if (measured <= tolerance && measured >= (-tolerance)) {
                nxt_motor_set_speed(MOTOR_STEERING, 0, 1);
            }
            else if (measured < (-tolerance)) {
                nxt_motor_set_speed(MOTOR_STEERING, speed, 0);
            }
            else if(measured > tolerance) {
                nxt_motor_set_speed(MOTOR_STEERING, -speed, 0);
            }
        break;
        case 1: // turn right
            if((degreeHist[0] < degreeHist[1] &&
                degreeHist[0] < degreeHist[2])) {
                nxt_motor_set_speed(MOTOR_STEERING, 0, 1);
            }
            else if(abs(degreeHist[0] - degreeHist[3]) < 3) {
                nxt_motor_set_speed(MOTOR_STEERING, 0, 1);
            }
            else if(degreeHist[0] < degreeHist[3]) {
                turn = 0;
            }
            else {
                nxt_motor_set_speed(MOTOR_STEERING, speed, 0);
            }
        break;
        case 2: // turn left
            ...
        break;
    }

    TerminateTask();
}
```

Code snippet 10.10. The body of the Wheels task.

saved in a variable `sonarRight`. There is an array, `rightHist`, which is a global variable containing the most recent history of the right sensor. `sonarRight` is then made room for in `rightHist` by moving the newest to the second newest and so forth until the oldest value has been overwritten with the second oldest value. `sonarRight`, the newest value, is then inserted at index 0 in `rightHist`, and the task is ended with `TerminateTask`.

The Distance Task

Code snippet 10.12 shows the structure of the **Distance** task, which was defined in Section 9.1.1. The snippet shows that there are two "states", defined by the `lockspeed` variable – introduced in Section 10.2. This variable is set to `1` if the Stalker is in the appropriate zone, and set to `0` if this is not the case. The naming of the appropriate zone was intuitive because it reiterates the need for the Stalker to hold its current speed if it is within the defined range. Otherwise it is allowed to accelerate or decelerate. Note that there is an exception to this which will be discussed later.

```
1  TASK(RightMeasure) {
2      int sonarRight = ecrobot_get_sonar_sensor(SONAR_RIGHT);
3
4      rightHist[3] = rightHist[2];
5      rightHist[2] = rightHist[1];
6      rightHist[1] = rightHist[0];
7      rightHist[0] = sonarRight;
8
9      TerminateTask();
10 }
```

Code snippet 10.11. The RightMeasure task.

The `if`-statement on line 10 puts the Stalker back into the state where deceleration/acceleration is allowed. In other words, it "unlocks" the *speedlock* if the Stalker is not in the appropriate zone – discussed in Section 9.1.1. Using the `filteredLeftHist` and `filteredRightHist` arrays, described in Section 10.2, it is verified that the Stalker is between two constants, the `brakeconst` and `accelconst` variables.

The second `if`-statement (line 19) shown in Code snippet 10.12, corrects the fault where the Stalker is outside the appropriate zone. Depending on the zone it is currently in, it will perform different computations.

```
1  int lockspeed = 0;
2
3  TASK(Distance)
4  {
5      int brakeconst = 15;
6      int accelconst = 25;
7
8      ...
9
10     if(filteredRightHist[0] > accelconst  ||
11         filteredLeftHist[0] > accelconst ||
12         filteredRightHist[0] < brakeconst  ||
13         filteredLeftHist[0] < brakeconst) {
14         lockspeed = 0;
15     }
16
17     ...
18
19     if(lockspeed != 1) {
20         ...
21     }
22
23     ...
24
25     TerminateTask();
26 }
```

Code snippet 10.12. The body of the Distance task.

Code snippet 10.13 shows the body of the `if`-statement found on line 19 in Code snippet 10.12. The `if`-statement on line 2 checks which one of the two sonar sensors are closest to the Runner. This approach is used because the shortest distance measured by one of the sensors is the one which tells how close the cars are to each other. In both

cases, almost identical computations are performed. The only difference is the name of the arrays and the usage of the greater-than/less-than signs (< and >). Therefore the code in the `else` clause – line 47 is omitted.

```
1  if(lockspeed != 1) {
2      if(filteredLeftHist[0] > filteredRightHist[0]) {
3          // acceleration zone
4          if(filteredRightHist[0] > accelconst && (approach == 1)) {
5              brake = 0;
6              currspeed -= ourRound(speedconstSlow);
7              mode = 1;
8          }
9          // acceleration zone
10         else if(filteredRightHist[0] > accelconst ) {
11             brake = 0;
12             currspeed += ourRound(speedconstAccel);
13             mode = 2;
14         }
15         // brake zone
16         else if ((filteredRightHist[0] < brakeconst) &&
17                 (filteredRightHist[0] >= stopconst) && approach == 0) {
18             brake = 0;
19             currspeed = 65;
20             mode = 3;
21         }
22         // brake zone
23         else if ((filteredRightHist[0] < brakeconst) &&
24                 (filteredRightHist[0] >= stopconst)) {
25             brake = 0;
26             currspeed -= ourRound(speedconstSlow);
27             mode = 3;
28         }
29         // stop zone
30         else if (filteredRightHist[0] < stopconst) {
31             currspeed -= ourRound(speedconstSlow);
32             brake = 1;
33             mode = 4;
34         }
35         // appropriate zone
36         else if (abs(filteredRightHist[0] - filteredLeftHist[0]) > 4) {
37             lockspeed = 1;
38         }
39         // appropriate zone
40         else if ((abs(filteredRightHist[3] - filteredRightHist[0]) == 0) &&
41                 ((filteredRightHist[0] < accelconst)  ||
42                 (filteredLeftHist[0] < accelconst))) {
43             lockspeed = 1;
44         }
45     }
46     else if(filteredLeftHist[0] <=  filteredRightHist[0]){
47         ...
48     }
49 }
```

Code snippet 10.13. The "locking" if-statement.

The first two checks – line 4 to 14 – are used to regulate the pace with which the Stalker accelerates, if the Stalker is in the accelerate zone. To explain this, an example is given. Assume that an error occurs which causes the Stalker to get further away from the Runner and out of the appropriate zone and into the acceleration zone. This will cause

both `filteredRightHist` and `filteredLeftHist` to be filled with values, which indicate that the Stalker is getting further and further away from the Runner. This would set the `lockspeed` variable's value to 0, and trigger the `if else`-statement on line 8, since `approach` would not be set to 1, as shown in Code snippet 10.14.

The `currspeed` would be increased (line 10 Code snippet 10.13), which would cause the Stalker to drive a bit faster, as it would increase the speed of the driving motor – seen on line 13 in Code snippet 10.15. Since the **LeftMeasure** and the **RightMeasure** tasks have a period of 50 ms and a higher priority than the **Distance** task, which has a period of 100 ms, both of the measuring tasks would be released twice before the **Distance** task would be released again. At this point, if the Stalker has increased its speed to a speed faster than that of the Runner, then both filtered arrays will be filled with values, indicating that the Stalker is getting closer to the Runner. This would cause `approach` to be set to 1 (line 14 in Code snippet 10.14) and the Stalker would decrease the `currspeed` variable on line 6 in Code snippet 10.13, which in turn would cause `approach` to be set to 0, causing the Stalker to increase `currspeed`.

This "increase-decrease" behavior would continue until `currspeed` is equal to the Runner's speed. This would cause `approach` to be set to 2 (line 21 in Code snippet 10.14), which would cause the Stalker to switch between a state where it travels slightly faster than the Runner, and a state where it travels with the same speed. Eventually the Stalker would enter the appropriate zone, and once again return to the state where `lockspeed` would be set to 1, and the fault which causes the Stalker to enter the acceleration zone to begin with would have been recovered.

The two `else if`-statements on line 16 to 28 in Code snippet 10.13, take care of the situation where the Stalker is in the brake zone (from Section 9.1.1). The intuition behind the desired behavior, is that the Stalker should only decrease `currspeed` as long as it can be decreased without making the Stalker stand still. Recall that 65 is almost the lowest possible driving speed, as described in Section 6.2. The second `else if`-statement in this context decreases `currspeed` (line 26), while the first `else if`-statement sets `currspeed` to the minimum PWM.

The next `else if`-statement on line 30 to 34 takes care of situations where the Stalker is in the stop zone. As defined in Section 9.1.1, the Stalker should stop the driving engine if it is in this state.

The last two `else if`-statements are to bring the Stalker back into the appropriate zone by setting the `lockspeed` variable to 1, and verifying that the needed conditions hold.

The `else if`-statement on line 36 to 37 takes care of situations where the Runner is turning more than the Stalker and locks the speed by setting `lockspeed` to `1`.

The final `else if`-statement on line 40 to 34 takes care of situations where the Stalker has travelled with the same speed as the Runner for a brief time, and the Stalker is not in the acceleration zone. In this state, the Stalker should enter the `lockspeed` state.

The first two `if`-statements in Code snippet 10.15, hold the `currspeed` variable within reasonable values because it makes no sense to have a higher `currspeed` than the maximum

```
int stopconst = 9;
int mode = 0;
int brake = 0;
float speedconstAccel = 0.50;
float speedconstSlow = 0.50;
int approach = 0;

// if we are turning
if (abs(degreeHist[0] - degreeHist[3]) > 3) {
    speedconstAccel = 0.25;
}

if((filteredRightHist[3] - filteredRightHist[0]) > 0 ) {
    approach = 1; // approaching
}
else if ((filteredRightHist[3] - filteredRightHist[0]) < 0){
    approach = 0; // opposite of approaching
}
else {
    approach = 2; // no change in distance
}

if(filteredRightHist[0] > accelconst ||
    filteredLeftHist[0] > accelconst ||
    filteredRightHist[0] < brakeconst ||
    filteredLeftHist[0] < brakeconst) {
    lockspeed = 0;
}

if(approach == 0 && lockspeed == 1) {
    currspeed += ourRound(speedconstAccel);
}
else if (approach == 1 && lockspeed == 1){
    currspeed -= ourRound(0.5);
}
```

Code snippet 10.14. Initial checks in the Distance task.

allowed (`maxspeed`) value.

```
    if (currspeed > maxspeed) {
        currspeed = maxspeed;
    }

    if (currspeed < 60) {
        currspeed = 60;
    }

    if (brake == 1) {
        nxt_motor_set_speed(MOTOR_DRIVE, 0, 1);
    }
    else {
        nxt_motor_set_speed(MOTOR_DRIVE, currspeed, 0);
    }

    TerminateTask();
}
```

Code snippet 10.15. The last lines of code in the Distance task.

11 Scheduling

Real-time systems must be schedulable in order to guarantee that all tasks always run at the chosen time and execute the desired operations. In order to determine whether the systems are schedulable or not, it is necessary to first gather data about the individual tasks, and then calculate utilization and a RTA.

11.1 Gathering Running Times

There are two kinds of running times to gather. To be able to account for blocking in the RTA running time, all critical sections of the code must be known. Secondly, the computation time of the tasks must be known for the RTA as well.

The following section will show the code used to gather these running times.

11.1.1 Assignment Test

In order to measure the time it takes to perform an assignment, the code, as shown in Code snippet 11.1, was run six times, with the results `15983` four times, `15984` one time, and `15985` one time.

```
1  int start = systick_get_ms();
2
3  for(int i = 1; i<= 100000000; i++) {
4   j = i;
5  }
6
7  int stop = systick_get_ms()+1;
8
9  display_goto_xy(0, 4);
10 display_string("Time:");
11 display_int(stop - start, 0);
12 display_update();
13
14 systick_wait_ms(40000);
```

Code snippet 11.1. Code used in the assignment test.

Using the largest observed running time and dividing it by the number of performed assignments:

$$\frac{15985ms}{100000000} = 0.15985\mu s$$

The above calculation is not just how long it takes to perform an assignment, but also all other operations performed in a single iteration of a `for`-loop. This means that the above time includes a lot of overhead, depending on how the compiler implemented the `for`-loop. Assuming that this number represents only the time needed for one assignment.

The critical sections in both the Runner and the Stalker is assignments of integers to a global variable from the sensors. Even with this overhead, the maximum blocking is still a short period of time. One could argue that such a short period is irrelevant, but it will be used as the maximum blocking time for both the Runner and Stalker.

11.1.2 Computation Times

The approach to gathering the computation times for the tasks is to run their body enough times to get a measurable time, as the lowest time unit is ms. The following approach was taken.

```
1  int i = 0;
2  int test_max;
3  int test_watch;
4  int starttime = systick_get_ms();
5
6  for (int j = 0; j < someint; j++){
7      ... Task program code ...
8  }
9  int endtime = systick_get_ms()+1;
10
11 test_watch[i] = (endtime-starttime);
12 ...
13
14 if(i >= 100000){
15     test_max = test_watch[0];
16     for(int i=1; i <= 100; i++){
17         if(test_watch[i] > test_max){
18             test_max = test_watch[i];
19         }
20     }
21
22     display_clear(0);
23     display_goto_xy(0, 0);
24     display_int(test_max, 0);
25     display_update();
26     systick_wait_ms(3000000);
27 }
28 i++;
```

Code snippet 11.2. The code used to measure the tasks' computation time, this example is for the Watchdog task.

In Code snippet 11.2, `someint` is the number of times it needs to run the function to get a measurable result in ms. The results for running the test-code on the Runner can be seen in Table 11.1. The task set for the Runner as seen in Table 11.2. The results for the Stalker can be seen in Table 11.3 and 11.4.

The smallest time unit in the system is 1 ms, thus all the tasks in the task set for the Runner and most of the tasks in task set for the Stalker have a computation time of 1 ms. One of the goals when gathering the running times, was to make the program take its longest code path. This was done by placing objects in front of the sensors to force the code to take a specific code path.

Task	Computation time, c
Watchdog	$\frac{8}{8000} = 1\mu s$
Wheel	$\frac{5}{1000} = 5\mu s$
Steering	$\frac{5}{100} = 50\mu s$

Table 11.1. Computation times for the Runner's task set.

Task	Period, T	Computation time, C	Deadline, D	Priority, P
Watchdog	5	1	5	3
Wheel	5	1	5	2
Steering	30	1	30	1

Table 11.2. The Runner's task set.

Task	Computation time, c
Watchdog	$\frac{72}{10000} = 7.2\mu s$
Wheels	$\frac{56}{10000} = 5.6\mu s$
Distance	$\frac{17}{10} = 1.7ms$
RightMeasure	$\frac{17}{100} = 0.17ms$
LeftMeasure	$\frac{17}{100} = 0.17ms$
Steering	$\frac{170}{100} = 1.7ms$

Table 11.3. Computation times the Stalker's task set.

Task	Period, T	Computation time, C	Deadline, D	Priority, P
Watchdog	5	1	5	6
Wheels	5	1	5	5
RightMeasure	50	1	50	4
LeftMeasure	50	1	50	3
Steering	100	2	100	2
Distance	100	2	100	1

Table 11.4. The Stalker's task set.

11.2 Utilization-based Schedulability Test

Following Section 2.3.4, a utilization-based schedulability test is conducted for the Runner and the Stalker's task sets. The calculations have been done with the following equation:

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq N \left(2^{1/N} - 1 \right)$$

This equation will, as explained in Section 2.3.4, sum all the utilizations of the tasks in a given task set and check if this is less than or equal to the right hand side. The right hand side is the utilization bound as previously explained.

Equation 11.1 uses the equation mentioned above to do the utilization-based schedulability test for the Runner. The data from Table 11.2 will be used.

$$\begin{aligned} \sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) &= \left(\tfrac{1}{5}\right) + \left(\tfrac{1}{5}\right) + \left(\tfrac{1}{30}\right) \\ &= 0.435 \\ N \left(2^{1/N} - 1 \right) &= 3 \left(2^{1/3} - 1 \right) \\ &= 0.78 \\ 0.435 &\leq 0.78 \end{aligned} \tag{11.1}$$

In Equation 11.1 it is apparent that it passes the utilization-based schedulability test, since $0.435 \leq 0.78$ is true. This means that with the current task set for the Runner it is schedulable.

In Equation 11.2 the data from Table 11.4 can be used in the same way as we just did for the Runner to do the test for the Stalker.

$$\begin{aligned} \sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) &= \left(\tfrac{1}{5}\right) + \left(\tfrac{1}{5}\right) + \left(\tfrac{1}{50}\right) + \left(\tfrac{1}{50}\right) + \left(\tfrac{2}{100}\right) + \left(\tfrac{2}{100}\right) \\ &= 0.48 \\ N \left(2^{1/N} - 1 \right) &= 6 \left(2^{1/6} - 1 \right) \\ &= 0.735 \\ 0.48 &\leq 0.735 \end{aligned} \tag{11.2}$$

As $0.48 \leq 0.735$ is true, it can be concluded that the Stalker is also schedulable, thus both of the cars have schedulable task sets.

11.3 Response Time Analysis

The utilization based test is a tool for an initial assessment on whether the program could be schedulable or not. It does have two major drawbacks, however. First of all, it is not

exact, meaning systems which fail a utilization based test might still be schedulable. The second is, that it is not applicable to a more general task model. Using a RTA instead allows for a more general approach by analyzing the WCRT (R) of each task. Comparing these values with each task's deadline allows for the tasks to be analyzed individually.

From Section 2.4 it is known that the computation time of every tasks is needed, before it is possible to conduct a RTA. These can be seen in Table 11.1 for both the Runner and the Stalker.

The purpose of the two following sections is to show whether they are schedulable using a RTA. This is built upon the theory in Section 2.4. The formula used to calculate the values are:

$$w_i^{n+1} = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{11.3}$$

which was explained in Section 2.4.

The following calculations will not take jitter into account. There are a couple of reasons for this. Since there is already an overhead on the cost of every task, it is deemed that the probability of jitter overextending this overhead is low. Furthermore, the calculations of the tasks' cost were calculated based on many run-throughs of the tasks. First, a loop executed the task a number of times. This loop was then executed a number of times, thus jitter would be represented in the task's cost calculations.

11.3.1 Runner

The task with the highest priority in the Runner is the **Watchdog** task with priority 3. The highest priority task's WCRT is equal to the sum of its computation time and maximum blocking, i.e. $R = B + C$. Recall the maximum blocking time from Section 11.1.1 which was calculated to be $B = 0.15985\mu s = 0.00015985ms$:

$$R = 0.00015985 + 1 = 1.00015985 \tag{11.4}$$

All other tasks will have some kind of interference from higher-priority tasks. Let w_{wheel}^0 be equal to the sum of its computation time and maximum blocking, $w_{wheel}^0 = 1ms + 0.15985\mu s$. Based in the calculation defined in 2.8, the WCRT for the **Wheels** task needs to be calculated:

$$\begin{aligned} w_{wheels}^0 &= 1.00015985 \\ w_{wheels}^1 &= 1.00015985 + \left\lceil \tfrac{1}{5} \right\rceil 1 = 2 \\ w_{wheels}^2 &= 1.00015985 + \left\lceil \tfrac{2}{5} \right\rceil 1 = 2 \end{aligned} \tag{11.5}$$

The value has now converged, and it can be seen that $w_{wheels}^1 = w_{wheels}^2 = 2$, therefore $R_{wheels} = 2$. The same procedure is used with the **Steering** task. First $w_{steering}^0$ is

set to be equal to the sum of its computation time and maximum blocking, $w^0_{steering} = 1ms + 0.15985\mu s$.

$$\begin{array}{lcl} w^0_{steering} & = & 1.00015985 \\ w^1_{steering} & = & 1.00015985 + \lceil \frac{1}{5} \rceil 1 + \lceil \frac{1}{5} \rceil 1 = 3 \\ w^2_{steering} & = & 1.00015985 + \lceil \frac{3}{5} \rceil 1 + \lceil \frac{3}{5} \rceil 1 = 3 \end{array} \tag{11.6}$$

The value has now converged, $w^1_{steering} = w^2_{steering} = 3$, therefore $R_{steering} = 3$.

Based on the values calculated in 11.4 (result: 1), 11.5 (result: 2) and 11.6 (result: 3), it can be concluded that all tasks meet their deadlines, as $R^w_{watchdog} \leq 2$, $R^w_{wheel} \leq 5$ and $R^w_{steering} \leq 30$.

11.3.2 Stalker

As with the Runner, the **Watchdog** task is the one with the highest priority in the Stalker.

Therefore, for the **Watchdog** task,

$$R = 0.00015985 + 0 + 1 = 1.00015985 \tag{11.7}$$

The **Wheels** task has second highest priority thus comes next:

$$\begin{array}{lcl} w^0_{wheels} & = & 0.00015985 + 1 \\ w^1_{wheels} & = & 0.00015985 + 1 + \lceil \frac{1}{5} \rceil 1 = 2.00015985 \\ w^2_{wheels} & = & 0.00015985 + 1 + \lceil \frac{2}{5} \rceil 1 = 2.00015985 \end{array} \tag{11.8}$$

The Stalker has two tasks to measure where to go, namely the **RightMeasure** task and the **LeftMeasure** task.

$$\begin{array}{lcl} w^0_{rightmeasure} & = & 0.00015985 + 1 \\ w^1_{rightmeasure} & = & 0.00015985 + 1 + \lceil \frac{1}{5} \rceil 1 + \lceil \frac{1}{5} \rceil 1 = 3.00015985 \\ w^2_{rightmeasure} & = & 0.00015985 + 1 + \lceil \frac{3}{5} \rceil 1 + \lceil \frac{3}{5} \rceil 1 = 3.00015985 \end{array} \tag{11.9}$$

$$\begin{array}{lcl} w^0_{leftmeasure} & = & 0.00015985 + 1 \\ w^1_{leftmeasure} & = & 0.00015985 + 1 + \lceil \frac{1}{5} \rceil 1 + \lceil \frac{1}{5} \rceil 1 + \lceil \frac{1}{50} \rceil 1 = 4.00015985 \\ w^2_{leftmeasure} & = & 0.00015985 + 1 + \lceil \frac{4}{5} \rceil 1 + \lceil \frac{4}{5} \rceil 1 + \lceil \frac{4}{50} \rceil 1 = 4.00015985 \end{array} \tag{11.10}$$

The **Steering** task has the same procedure, only with all its higher-priority tasks in the calculation as well.

$$\begin{aligned} w^0_{steering} &= 0.00015985 + 2 \\ w^1_{steering} &= 0.00015985 + 2 + \left\lceil \tfrac{2}{5} \right\rceil 1 + \left\lceil \tfrac{2}{5} \right\rceil 1 + \left\lceil \tfrac{2}{50} \right\rceil 1 + \left\lceil \tfrac{2}{50} \right\rceil 1 = 6.00015985 \\ w^2_{steering} &= 0.00015985 + 2 + \left\lceil \tfrac{6}{5} \right\rceil 1 + \left\lceil \tfrac{6}{5} \right\rceil 1 + \left\lceil \tfrac{6}{50} \right\rceil 1 + \left\lceil \tfrac{6}{50} \right\rceil 1 = 8.00015985 \\ w^3_{steering} &= 0.00015985 + 2 + \left\lceil \tfrac{8}{5} \right\rceil 1 + \left\lceil \tfrac{8}{5} \right\rceil 1 + \left\lceil \tfrac{8}{50} \right\rceil 1 + \left\lceil \tfrac{8}{50} \right\rceil 1 = 8.00015985 \end{aligned} \tag{11.11}$$

And finally, the same procedure is used for the **Distance** task.

$$\begin{aligned} w^0_{distance} &= 0.00015985 + 2 \\ w^1_{distance} &= 0.00015985 + 2 + \left\lceil \tfrac{2}{5} \right\rceil 1 + \left\lceil \tfrac{2}{5} \right\rceil 1 + \left\lceil \tfrac{2}{50} \right\rceil 1 + \left\lceil \tfrac{2}{50} \right\rceil 1 + \left\lceil \tfrac{2}{100} \right\rceil 2 = 8.00015985 \\ w^2_{distance} &= 0.00015985 + 2 + \left\lceil \tfrac{8}{5} \right\rceil 1 + \left\lceil \tfrac{8}{5} \right\rceil 1 + \left\lceil \tfrac{8}{50} \right\rceil 1 + \left\lceil \tfrac{8}{50} \right\rceil 1 + \left\lceil \tfrac{8}{100} \right\rceil 2 = 10.00015985 \\ w^3_{distance} &= 0.00015985 + 2 + \left\lceil \tfrac{10}{5} \right\rceil 1 + \left\lceil \tfrac{10}{5} \right\rceil 1 + \left\lceil \tfrac{10}{50} \right\rceil 1 + \left\lceil \tfrac{10}{50} \right\rceil 1 + \left\lceil \tfrac{10}{100} \right\rceil 2 = 10.00015985 \end{aligned} \tag{11.12}$$

Comparing with the periods in Table 11.4, the results from Equation 11.7, Equation 11.8, Equation 11.9, Equation 11.10, Equation 11.10 and Equation 11.10 shows:

$$\begin{gathered} 1.00015985 \leq 5 \\ 2.00015985 \leq 5 \\ 3.00015985 \leq 50 \\ 4.00015985 \leq 50 \\ 8.00015985 \leq 100 \\ 10.00015985 \leq 100 \end{gathered}$$

These results show that all tasks in the Stalker will respond within their respective deadlines.

Therefore the conclusion for both the Runner and Stalker is that they are schedulable.

Fault Handling 12

In this chapter the fault handling implemented will be tested and evaluated. The faults that need handling is first the fault where the Stalker leaves the appropriate zone as described in section 9.1.1, and secondly the fault where the tasks exceed their WCET.

12.1 Distance Faults

Four distance zones have been defined for the Stalker. The goal is to maintain a distance from 15 and 25 cm between the two cars. There are three other zones than the appropriate zone, and the error handling must be tested for all three zones.

12.1.1 The Stop Zone

The stop zone is defined in section 9.1.1 to be the zone where the distance between the cars is 0-9 cm.

To test if the Stalker is able to handle the fault of entering the stop zone, two tests were performed. The purpose of the tests was to confirm that the Stalker stops before crashing into the Runner, and that it is able to reenter the correct zone from having stopped.

To document that the Stalker stops before crashing when entering the stop zone, it was set to drive towards a wall with its maximum velocity, which is a `PWM` value of `85`. The test is designed to test the worst possible scenario, i.e. the Stalker driving towards a stationary target with its maximum velocity. The Stalker should stop before hitting the wall. The test was performed five times and the distance between the wall and the Stalker was measured and is shown in Table 12.1.

Test nr.	Distance
1	0.8 cm
2	0.9 cm
3	0.6 cm
4	1.5 cm
5	0.7 cm

Table 12.1. Stop test results

As the test data shows the Stalker stops with a distance of less than 1 cm to the wall in

all cases except one. In some tests the Stalker stopped, and due to the looseness of the construction and the forward momentum, it bounced slightly on the wall before locking into the stop position. The case where the measurement was larger than 1 cm, the Stalker turned slightly before detecting the wall. This caused it to readjust its direction towards the wall, resulting in a smoother approach.

In general when approaching with maximum velocity towards a stationary target it is capable of stopping just in time. In realistic circumstances however it will be driving towards a moving target meaning it will stop with a safe distance to the Runner, should it enter the stop zone, documented in [13].

To document that the Stalker is able to reenter the correct zone the cars must be set to drive a number of times, and it must be observed that the Stalker is able to reenter the appropriate zone in case it leaves it.

The test case involved the Stalker driving too fast towards the Runner, hereby entering the stop zone. The test showed that the Stalker did stop as expected. However it was noted the overall behavior of the Stalker when entering the braking zone depended on the circumstances under which it occurred.

A case where the Stalker does not recover correctly is when the Runner makes a sharp turn, and the Stalker begins its turn, before entering the braking zone. This locks the Stalker's wheels in a sharp turn position, and if the Runner then drives in a straight line it is not always detected by the Stalker. This is however not an issue when the Runner drives smoothly.

Another case is when the Stalker approaches the Runner with a high velocity. The Stalker first drives through the brake zone and enters the stop zone. If the initial `PWM` value is the maximum value of 85, it will not be slowed down enough to stabilize its velocity in the brake zone. Therefore it enters the stop zone, and correctly stops, and as the Runner drives away it starts moving again and finds the appropriate zone.

These are the only cases where the Stalker enters the stop zone, and in most cases it is able to recover and find the appropriate zone. The error handling for the stop zone is deemed acceptable as the only faults that occur is in cases which would not exist in a real world scenario.

12.1.2 The Brake Zone

The Stalker enters the brake zone when its velocity is higher than the velocity of the Runner. As mentioned in the previous section there are cases where the Stalker drives through the brake zone and into the stop zone. These cases are described and they will not be covered in this section.

The case where the Stalker is driving with a velocity so high that the measurements taken in the appropriate zone are not sufficient enough to detect that the Stalkers speed must be decreased, and therefore it enters the brake zone. There are two outcomes in this scenario: First, if the velocity of the Stalker is too high it will enter the stop zone. This case was covered in the previous section. Second, if the velocity of the Stalker is low enough, it will

be decelerated enough in the brake zone to avoid entering the stop zone.

12.1.3 The Acceleration Zone

A fault occurs when the Stalker does not detect that its velocity is lower than the velocity of the Runner. This can occur when the Runner turns too sharply and the Stalker then assumes that the Runner is very far away, because it cannot detect the surface on the Runner.

To ensure that the Stalker acted according to the desired behavior the cars were run a number of times to observe the actual behavior.

It was observed that when the Stalker detected the Runner properly it correctly entered the appropriate zone, and when it had not detected it properly there were cases where it lost track of the Runner during sharp adjustments.

12.1.4 Evaluation

In general the Stalker is able to recover from many faults which occurred when running. The cases which cause trouble are when the Runner performs quick sharp adjustments, and the Stalker has not properly detected the Runner. However, as the main purpose of the project is to follow it is deemed acceptable, as it recovers from all errors which occur when the Stalker has detected the Runner.

12.2 WCET Overrun

Faults might occur in the system of both the Runner and the Stalker which might cause tasks to exceed their WCET. As it cannot be detected whether the task exceeded its WCET due to a fault e.g. jitter, it must be assumed a fault has occurred in the system, and an action will need to be taken to recover from it. As described, this is handled by the **Watchdog** task, which monitors the running time of all other tasks in the system, and detects when they exceed their WCET. The **Watchdog** task must be tested to ensure that the fault handling works as intended.

12.2.1 WCET Faults

To test the fault handling of the **Watchdog** task, the code of the **Steering** task was modified to ensure it exceeded its measured WCET in both cars. It was modified by adding a `for`-loop performing a mathematical calculation. The fault handling for the **Watchdog** task is a loud sound, which is played to alert the driver of the car that the system needs rebooting.

The Stalker was run and as the **Steering** task was run it exceeded its WCET. It was correctly detected by the system that the WCET was exceeded and the **Watchdog** task performed the desired action to alert the driver of the fault.

The Runner was run and the same result as in the Stalker was observed.

As the same code is being used to monitor all tasks it is deemed redundant to repeat the same test for all tasks, and it was concluded that the **Watchdog** task and the associated error handling works as intended.

12.3 Conclusion

Within the set boundaries of the system the fault handling works as intended. The main purpose of the project is to get the Stalker to follow the Runner. When the Stalker has detected the Runner properly it recovers correctly from faults. Furthermore most faults stems from scenarios, which would not happen in a real-life scenario on a highway, such as sharp turns or adjustments made by the Runner.

Evaluation of Implementation 13

A lot of the difficult decisions, such as which hardware to use or which operating system to use were made in Part II and III. Everything that was designed in that part was implemented in this part.

As the process of implementation progressed, some design flaws became apparent. Specifically, programming the Stalker to follow the Runner in a smooth line proved to be more difficult to implement than originally anticipated. Therefore tests were performed to evaluate the performance of the implementation.

13.1 Testing the Implementation

Two concluding tests were conducted at the end of the two cars' development. The first test was designed to test how well the Stalker would run in a straight line. The second test was intended to test how well the Stalker would make a turn while following the Runner. Since it is difficult to measure accurately if the Stalker is following a line behind the Runner the tests relied on video recording the two cars in action.

Before concluding on the tests it is important to note that the Runner is designed to drive randomly while avoiding any obstacle in its way, such as a wall, enabling it to make sharp turns up to 180 degrees.

The results were quite satisfying for the first test as the car kept within half a car's distance on either side of the Runner, thus keeping it very close to the line the Runner was driving. The test video is found in [17].

The results of the second test were not quite as satisfying as those of the first test. As anticipated, it was more difficult than the straight line. It made the turn, but it was not effortless and quite a few times it nearly lost sight of the Runner. The Stalker did make the turn, however, and thus we can be satisfied with this test as well. The test video is found in [18].

Two additional tests were performed to test specific parts of the system.

A test was performed to test the dynamic detection of the Stalker i.e. where the cars are not started behind each other. The test showed the Stalker eventually detected the Runner

properly, however it had some difficulties locking itself into the appropriate distance zone. The difficulties arose from the Stalker approaching when a sharp turn was made by the Runner causing the Stalker to turn away. The test video is found in [12].

Another test was performed to test the ability of the Stalker to stop before crashing into the Runner. Beside the static test done in Section 12.1.1, a dynamic test was performed. It was noted the Stalker stopped before crashing into the Runner when it was suddenly stopped. The test video is found in [13]

Many smaller tests were also conducted during the development. Different values were adjusted based on eye judgments to make the Stalker behave more desirably, i.e. driving in a smoother path behind the Runner. The values and constants that were changed during the development are as follows:

- Distance to the Runner
- Acceleration and deceleration constants
- Acceleration and distance tolerances
- Turning-speed constants
- Turning-tolerance

All these values were adjusted over a period of three weeks. We found this to be the best way to test the implementation of the project. As it has already been mathematically proven in Section 11.3 that the two cars are schedulable and that all tasks will meet their deadlines. This was only enhanced by the fact that the watchdog did not react during any of the tests.

Some difficulties were experienced while using the two sonar sensors on the Stalker. For example, it exhibited some strange behaviors such as turning more to one side than the other. This made it difficult to get the desired result since it was not immediately solvable by software, but instead relied on a different hardware approach. While testing we realized that, due to the design of the sonar sensors, the right sonar sensor would receive signals from the left sonar sensor, but not vice versa. This occurs because the sonar sensors transmit with the right "head" and receive with the left "head" as explained in Section 3.4. To counter this interference we implemented a feature that shuts down the sensors when they are not being used. This only had a limited effect, however, as the sound waves were still floating in the air. This meant that the measurements, if too frequent, would still be of the old sound waves. To compensate for this, a filter was implemented, that keeps a history of previous measurements. If a faulty measurement was observed, it was discarded and the last valid measurement was used instead.

13.2 Result of the Implementation

Due to the interference experienced by the sensors on the Stalker, it is not possible to run at maximum speed, since one incorrect measurement could cause the Stalker to suddenly make a sharp turn and lose sight of the Runner. When the Stalker drives at lower speeds this issue is less significant, since even a sharp turn will not cause it to turn fast enough to lose sight of the Runner.

Another issue with speed arises if the Runner drives at the Stalker's max speed or above because it is almost impossible for the Stalker to follow. This is connected to the other problem with speed as the Stalker sometimes performs seemingly random turns, and will be turning both randomly and when the Runner turns. This is not the only problem with high speed on the Runner, however, as it is programmed to avoid obstacles. Therefore it may make some sudden turns, e.g. if it sees a wall ahead it will start making a turn to the left.

Since the Stalker is implemented with two distance sensors trying to follow a surface on the Runner it will lose sight of the surface if the Runner suddenly makes a sharp 90 degree turn. This is because the surface becomes a very thin line impossible for the Stalker to detect. This is not very likely on the highway, however, as the road will be straight.

Due to the aforementioned issues, the current implementation works better when driving at lower speeds. It is worth mentioning that if the Runner was driving in a straight line, it would be no problem for the Stalker to follow it, even if it was driving faster.

It is implemented this way to account for any possible scenario that might occur on the highway. The tests show that an implementation using distance sensors works best when driving in as straight a line as possible. This is expected of a real world highway implementation of the system.

It is plausible that an implementation using the IR-sensors could have handled the sharp turns better than the current implementation. That technology is not available, however. We proved through the tests that using sonar sensors is a viable solution when used in a highway scenario.

Part V

Epilogue

Closing chapter 14

The overall purpose of this project was, through designing a real-time system, to gain knowledge of the concepts and considerations involved in this process. While the structure of the process of designing a real-time system is similar to that of designing a desktop application, in the sense both have an analysis and a design phase, the work done in the phases differs from each other. A large part of the analysis concerning real-time systems, in this project, involved testing and evaluating the available hardware, since it limits what is possible to create. The design phase also involved dividing functionality into tasks compared to classes or methods. The most significant difference is, however, that a large part of designing a system involves non-code related areas such as scheduling and RTA.

The main difference from constructing a real-time system compared to a regular system is the hardware aspect. When programming desktop applications, the computer can to some extent be considered a black box, which is given a piece of code and returns a result. For real-time systems this is not the case. Hardware components are actively used in the software to provide data, or are programmed to perform an action.

During the project, it was discovered that the quality and reliability of the hardware provided a limitation. It was simply unreliable. The approach taken in this project was to find a problem to solve, and then test the hardware according to the chosen problem. When limited quality hardware was available, we noticed this could cause a problem. We found that while the constructed system did fulfill the main properties defined in the problem statement, the initial expectation were higher before the hardware was properly tested. Therefore a better approach for a project such as this, would be to examine the hardware first, and then, based on the results, choose the project.

Conclusion

The research question for the project was:

How can we design and implement a road-train system consisting of one leader vehicle and one follower vehicle, using LEGO NXT?

Using the LEGO NXT, an embedded real-time system was programmed using the C language on the nxtOSEK platform. The system consisted of two vehicles constructed with LEGO. One car was made with the purpose of leading the other car safely. The

other was made to follow the leader car.

To achieve this functionality sonar sensors were used on both cars. The Runner used sonar sensors to detect obstacles and avoid them, while the Stalker used sonar sensors to detect a surface on the Runner and, based on the received input, follow it.

The hardware setup was chosen based on tests conducted on the available hardware. The purpose of the tests was to identify the hardware's properties, and then evaluate the possible hardware setups with these in mind.

For controlling the Stalker, a dynamic approach was taken, where it reacts on what it has observed over a period of time. If it has observed the Runner is turning, it turns. If it has observed it is turning with the same angle as the Runner, it locks the wheels. If the Runner stops turning, it stops turning as well.

A proper distance to the Runner is being kept by dividing the distances into zones, and based on the distance zone the Stalker is in, it either stops, decelerates, accelerates or maintains the speed.

The costs of the implemented tasks have been measured, and using these the system has been tested for schedulability. Both the utilization-based test and the RTA confirmed that the system was schedulable.

Testing the implementation documented that in most cases, the system handled errors correctly. Distance faults were recovered from, and when WCET overruns occurred, the system correctly notified the user of the fault. However, conducting a test that covers all possible scenarios can not be done within the boundaries of the project, as an autonomous system will behave differently depending on which environment it is run in.

To summarize; the designed systems was build with LEGO and was designed as defined: One to lead, one to follow. They were tested and analyzed which confirmed that they were schedulable, and in most cases handled errors correctly. If any faults occurred, the user would be notified. Even though there are still areas in the system which could be improved, the overall functionality conforms with the properties presented in Section 1.2.

Future work 15

The project intended to uncover some of the problems with constructing a road-train through constructing a system which emulates the scenario. Despite the project being on such a small scale it demonstrated the importance of having quality hardware available.

Should the project be developed further, some of the algorithms used should be improved to be more dynamic. The current distance algorithm is static and a dynamic one could possibly improve its ability to maintain the speed of the Runner. Furthermore in a real-world scenario with time to examine and handle the problems with exchanging data between the two cars, this would probably have been a more viable approach, as it removes the dependency on sensors. In this scenario the Stalker would receive information on what velocity it should be driving with, and adjust accordingly. However, in this project with the unpredictable movements of the Runner, controlling the speed of the Stalker statically seemed better.

The steering algorithm in the Stalker makes use of the sonar sensors as it was the most viable of the working hardware. With better hardware, a better steering algorithm could be implemented. The initial idea in the project was to use IR to detect the Runner. As nxtOSEK does not support the IRSeeker V2 this approach could not be used. This idea could not be investigated to the fullest since it can not be determined if using the IRSeeker V2 would have resulted in a better follow algorithm for the Stalker. With time to implement a working driver for the IRSeeker V2 the approach should be tested more thoroughly.

For the implemented solution an error was found on line four in Code snippet 10.1, where a `systick_wait_ms(200);` statement had been omitted. However, as the system had been implemented in its entirety with this error, it was not corrected, as it affected the functionality of the entire system if it had been corrected. With time available the system should be changed to work without this error.

Bibliography

[1] Josie Garthwaite. Eu to trial "road train" tech: Sensor-enabled automated driving (whoa!). *URL: http://gigaom.com/cleantech/eu-to-trial-road-train-tech-sensor-enabled-automated-driving-whoa/*, 2009. Last viewed: 2011-12-07.

[2] René Rydhof Hansen. Real-time software - basic scheduling. *URL: https://intranet.cs.aau.dk/uploads/media/lecture04_01.pdf*, 2009.

[3] Martin Hiller. Software fault-tolerance techniques from a real-time systems point of view - an overview. *URL: https://www.cs.drexel.edu/~bmitchel/course/cs575/ClassPapers/hiller98software.pdf*, 1998. Last viewed: 2011-12-07.

[4] HiTechnic. Hitechnic nxt irseeker v2 sensor for lego mindstorms nxt. *URL: http://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=NSK1042*, 2001-2009. Last viewed: 2011-12-07.

[5] HiTechnic. *HiTechnic IR RC Car*. HiTechnic, 2009.

[6] leJOS. lejos documentation. *URL: http://lejos.sourceforge.net/*, 1997-2009. Last viewed: 2011-12-07.

[7] Brian Randel & Jean-Claude Laprie & Hermann Kopetz & Bev Littlewood. *Predictably Dependable Computing Systems*. Springer, 1st, edition, 1995.

[8] air resistance Mythbusters. Mythbusters prove air resistance when following a big truck. *URL: http://green.autoblog.com/2007/10/28/mythbusters-drafting-10-feet-behind-a-big-rig-will-improve-mile/*, 2007. Last viewed: 2011-12-07.

[9] nxtOSEK. nxtosek rate monotonic scheduling. *URL: http://lejos-osek.sourceforge.net/rms.htm*, 2007-2010. Last viewed: 2011-12-07.

[10] nxtOSEK. nxtosek event driven scheduling. *URL: http://lejos-osek.sourceforge.net/eds.htm*, 2011. Last viewed: 2011-12-07.

[11] OSEK. Osek/vdx operating system specification 2.2.3. *URL: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf*, 2005. Last viewed: 2011-12-07.

[12] Anders Eiler & Thomas Panum. Implementation test 3 - detection and general following. *URL: http://vimeo.com/33340172*, 2011. Last viewed: 2011-12-14.

[13] Anders Eiler & Thomas Panum. Implementation test 4 - stop test while driwing. *URL:* *http://vimeo.com/33340466*, 2011. Last viewed: 2011-12-14.

[14] Open Source Physics. Tracker - video analysis and modeling tool. *URL:* *http://www.cabrillo.edu/~dbrown/tracker/*, 2012. Last viewed: 2011-12-07.

[15] Divya Kumari Prasad. Dependable systems integration using measurement theory and decision analysis. *URL:* *http://www-users.cselabs.umn.edu/classes/Spring-2009/csci5980-med/YCST-99-02.pdf*, 1998. Last viewed: 2011-12-12.

[16] Ola Redell and Martin Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. *URL:* *http://www.artes.uu.se/events/gsconf02/papers/redell.pdf*, 2002. Last viewed: 2011-12-07.

[17] Bjarke H. Søndergaard & Magnus S. Reichenauer. Implementation test 1 - straight line. *URL:* *http://vimeo.com/33650512*, 2011. Last viewed: 2011-12-14.

[18] Bjarke H. Søndergaard & Magnus S. Reichenauer. Implementation test 2 - turning. *URL:* *http://vimeo.com/33650323*, 2011. Last viewed: 2011-12-14.

[19] TOPPERS. Toppers atk1 api. *URL:* *http://lejos-osek.sourceforge.net/ecrobot_c_api.htm*, 2011. Last viewed: 2011-12-07.

[20] Alan Burns & Andy Wellings. *Real-Time Systems and Programming Languages.* Addison Wesley, 4th, edition, 2009.

[21] Wikipedia. Byzantine fault tolerance. *URL:* *http://en.wikipedia.org/wiki/Byzantine_fault_tolerance*, 2011. Last viewed: 2011-12-07.

Appendix

LEGO NXT Hardware Tests A

A.1 Scope Test

The purpose of this test is to determine the scope of the sonar sensors measuring area. The test was performed using:

- Table of the dimentions 120 cm x 80 cm, placed against a wall.
- NXT running a nxtOSEK program, which displays the current measurement of the sonar sensor.
- Obstacle, in this case a cardboard box of the dimentions: 15 cm x 6 cm x 3,5 cm.

The test setup is illustrated in Figure A.1. The sonar sensor is placed in the center, and with no obstacles it measures ≈ 120. When the output of the sensor is ≈ 120, it is assumed that the sensor detects the wall. The test is conducted by placing the box object at specific x-coordinate and then starting from one side of the table adjusting the y-coordinate until the sonar sensor dectects the box i.e. giving a stabilized non-120 output.

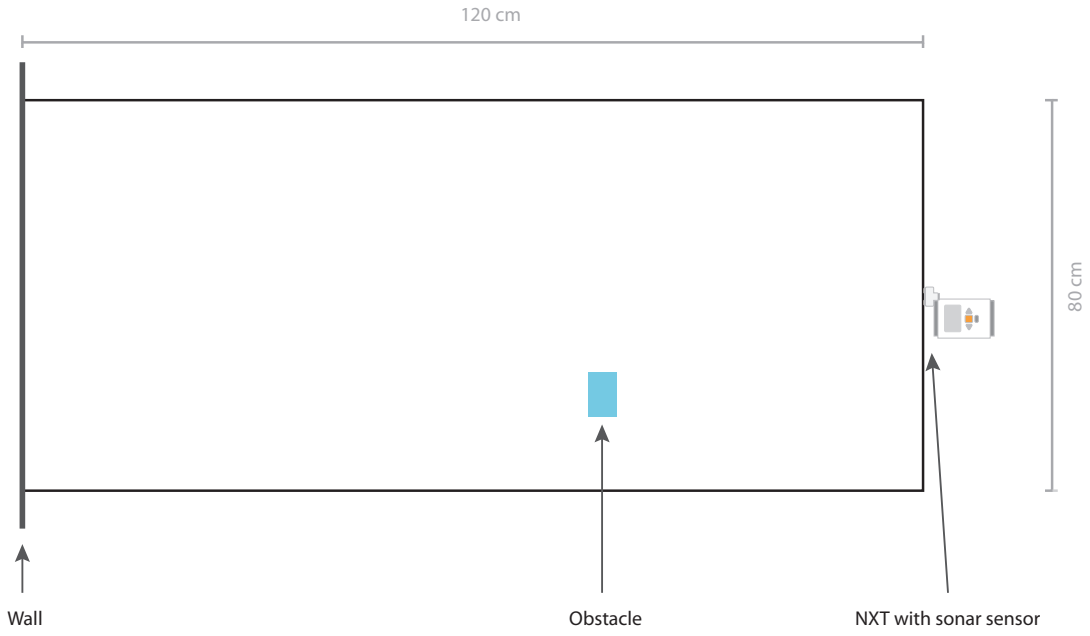

Figure A.1. Scope Test setup

The results of the test is the edge of the scope, i.e. the area between the lines is the scope. The results is shown in Figure A.2.

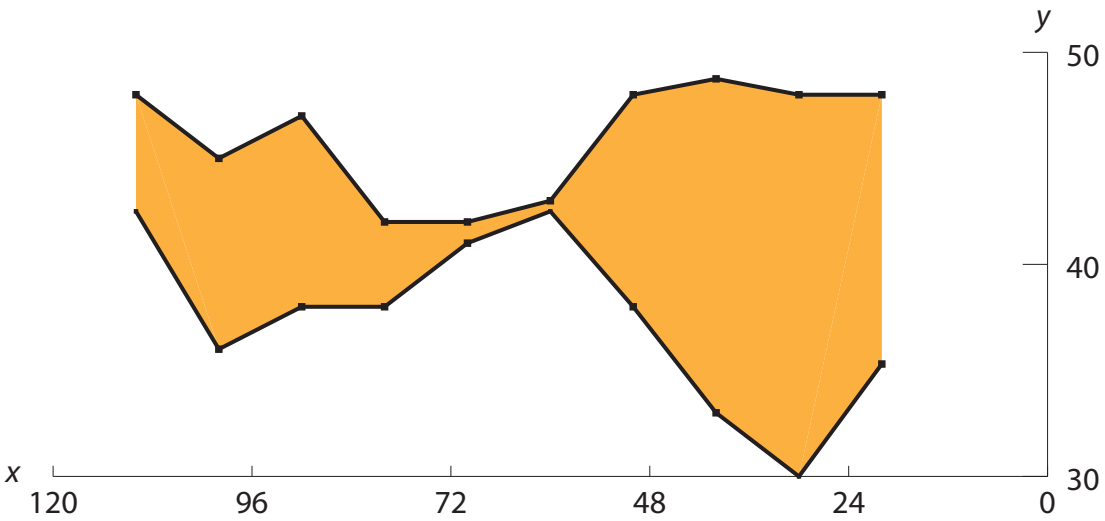

Figure A.2. Scope Test results

A.2 Motor Test

PWM	RPM
20	30
40	62
60	91
80	122
100	163

Table A.1. Data for speed test of LEGO NXT Motor 1.

PWM	RPM
20	30
40	61
60	90
80	121
100	161

Table A.2. Data speed test of LEGO NXT Motor 2.

A.3 Sonar Test

The raw data from the distance test of the LEGO NXT Sonar Sensor is shown in Table A.6 and A.6. The test was performed three times for each sensor, which gave the average data used in section 3.4. The test was done by measuring some distances with a sort of ruler and then place the sensors at these measured distances and read the input they give.

Reference Distance	Measured Distance	Relative Deviation
5	7	28.57%
10	13	23.08%
20	22	9.09%
40	40.33	0.83%
60	60	0%
80	80	0%
100	100	0%
125	125	0%
150	150	0%
175	175	0%
185	185	0%
200	201	0.5%

Table A.3. Distance test for LEGO NXT Sonar Sensor 2.

Reference Distance	Measured Distance	Relative Deviation
5	7	28.57%
10	13	23.08%
20	20	0%
40	40.33	0.83%
60	60	0%
80	80	0%
100	100	0%
125	125.33	0.27%
150	150	0%
175	175.67	0.38%
185	186.33	0.76%
200	201.67	0.83%

Table A.4. Distance test for LEGO NXT Sonar Sensor 3.

Reference Distance	Measured Distance	Relative Deviation
5	7	28.57%
10	12	16.67%
20	22	9.1%
40	40.67	1.64%
60	61	1.64%
80	80.33	0.41%
100	100	0%
125	125.33	0.27%
150	150.33	0.22%
175	175	0%
185	186	0.54%
200	200.67	0.33%

Table A.5. Distance test for LEGO NXT Sonar Sensor 5.

Reference Distance	Sonar 1	Sonar 2	Sonar 3	Sonar 4	Sonar 5
5	8	7	7	7	7
10	13	13	13	12	12
20	23	22	20	21	22
40	41	40	41	40	40
60	61	60	60	60	61
80	81	80	80	80	80
100	101	100	100	100	100
125	126	125	126	125	125
150	152	150	150	150	150
175	178	175	175	175	175
185	188	185	186	185	186
200	255	200	202	200	200
5	7	7	7	7	7
10	12	13	13	12	12
20	23	22	20	21	22
40	41	40	40	40	41
60	61	60	60	60	61
80	81	80	80	80	81
100	101	100	100	100	100
125	126	125	125	125	126
150	152	150	150	150	151
175	178	175	177	175	175
185	255	185	187	185	186
200	255	202	202	200	201
5	7	7	7	7	7
10	13	13	13	12	12
20	23	22	20	21	22
40	41	41	40	40	41
60	61	60	60	60	61
80	81	80	80	80	80
100	101	100	100	100	100
125	126	125	125	125	125
150	151	150	150	150	150
175	177	175	175	175	175
185	188	185	186	185	186
200	255	201	201	200	201

Table A.6. Raw data from our LEGO NXT Sonar Sensor distance test.

A.4 Brake Length Test

```
/* nxtOSEK hook to be invoked from an ISR in category 2 */
void user_1ms_isr_type2(void){ /* do nothing */ }

TASK(OSEK_Task_Background)
{
    nxt_motor_set_speed(MOTOR_DRIVE,65,0);
    systick_wait_ms(2000);
    ecrobot_sound_tone(2000,1000,100);
    nxt_motor_set_speed(MOTOR_DRIVE,0,1);
    systick_wait_ms(5000);
    while (1) {
        ecrobot_sound_tone(500,1000,20);
        systick_wait_ms(2000);
    }
}
```

Code snippet A.1. Brake Length Test - Source code

A.4.1 Brake Length Raw Data

First value is the last point recorded before the car starts braking.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
1.066666667	35.57434662402330	47.20442148187710	
1.1	35.80238730751060	47.43246216536440	0.322498227
1.133333333	35.80238730751060	49.14276729151940	1.710305126
1.166666667	36.71455004145990	50.96709275941800	2.03965788
1.2	37.17063140843460	53.13347925254760	2.213874579
1.233333333	37.62671277540930	54.95780472044630	1.88047165
1.266666667	37.28465175017830	54.95780472044630	0.342061025
1.3	37.74073311715290	56.32604882137020	1.442255918
1.333333333	37.74073311715290	56.55408950485760	0.228040683
		Break Distance	8.146361735

Table A.7. Brake test 1 at 100 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
0.133333333	2.860143815	17.47865665	
0.166666667	2.754212563	20.02100671	2.544556002
0.2	2.860143815	20.12693796	0.149809414
0.233333333	2.754212563	22.45742551	2.332893839
0.266666667	2.966075068	24.15232555	1.708090121
0.3	2.860143815	26.05908809	1.909702811
0.333333333	2.754212563	27.22433187	1.170048926
0.366666667	2.754212563	27.22433187	0
0.4	2.754212563	27.75398813	0.529656262
0.433333333	2.966075068	27.75398813	0.211862505
		Break Distance	8.012063879

Table A.8. Brake test 2 at 100 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
0.266666667	9.232365145	22.406639	
0.3	9.232365145	25.5186722	3.112033195
0.333333333	9.439834025	27.48962656	1.98184369
0.366666667	9.647302905	29.46058091	1.98184369
0.4	9.543568465	31.63900415	2.180891706
0.433333333	9.543568465	31.63900415	0
0.466666667	9.751037344	32.67634855	1.057887866
0.5	9.647302905	33.29875519	0.630991964
0.533333333	9.647302905	33.60995851	0.31120332
0.566666667	9.647302905	33.50622407	0.10373444
		Break Distance	8.248396675

Table A.9. Brake test 3 at 100 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
1.066666667	10.13386384	46.02229879	2.139306043
1.1	9.909911052	47.81392112	1.805565134
1.133333333	9.909911052	49.04566148	1.231740357
1.166666667	10.02188745	50.94926021	1.906889313
1.2	10.02188745	50.72530742	0.223952792
1.233333333	9.909911052	51.3971658	0.681125826
1.266666667	9.909911052	51.84507138	0.447905584
		Break Distance	3.259873515

Table A.10. Brake test 1 at 85 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
0.233333333	9.19878943	19.58801043	1.637681688
0.266666667	9.307010482	21.31954727	1.734915446
0.3	9.19878943	22.94286305	1.626919151
0.333333333	9.090568378	23.0510841	0.15304768
0.366666667	9.19878943	23.80863147	0.765238398
0.4	9.307010482	24.24151567	0.446206829
0.433333333	9.19878943	24.34973673	0.15304768
0.466666667	9.307010482	24.56617883	0.241989629
0.5	9.090568378	24.45795778	0.241989629
		Break Distance	3.628438995

Table A.11. Brake test 2 at 85 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
0.5	10.64066339	22.25110877	1.885687184
0.533333333	10.58678662	24.0290424	1.778749758
0.566666667	10.58678662	23.97516562	0.053876777
0.6	10.53290984	25.32208504	1.347996522
0.633333333	10.69454017	26.13023669	0.824156207
0.666666667	10.69454017	26.50737412	0.377137437
0.7	10.69454017	26.50737412	0
0.733333333	10.64066339	26.50737412	0.053876777
0.766666667	10.58678662	26.23799024	0.274718736
0.8	10.64066339	25.91472958	0.327719638
0.833333333	10.64066339	25.80697603	0.107753553
0.866666667	10.64066339	25.53759214	0.269383883
		Break Distance	3.582742752

Table A.12. Brake test 3 at 85 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
0.533333333	14.33217152	21.93256551	
0.566666667	14.22359447	22.80118197	0.875376219
0.6	14.11501741	24.32126077	1.523951616
0.633333333	14.33217152	25.29845428	1.001031004
0.666666667	14.44074858	25.18987722	0.153551147
0.7	14.33217152	25.51560839	0.343350802
0.733333333	14.22359447	26.27564779	0.767755733
0.766666667	14.54932564	26.60137896	0.46065344
0.8	14.44074858	25.40703134	1.199272793
0.833333333	14.44074858	25.29845428	0.108577057
0.866666667	14.54932564	25.29845428	0.108577057
		Break Distance	4.142769032

Table A.13. Brake test 1 at 75 PWM.

t	x	y	$\|\overrightarrow{xy}\|$
0.7	9.762375857	-22.83052978	
0.733333333	9.762375857	-22.778877	0.051652782
0.766666667	9.81402864	-23.96689099	1.18913635
0.8	9.865681422	-25.30986333	1.343965296
0.833333333	9.917334204	-26.44622454	1.137534532
0.866666667	9.865681422	-27.06605793	0.621981863
0.9	9.968986987	-27.01440515	0.115499132
0.933333333	9.917334204	-27.32432184	0.314191609
0.966666667	9.917334204	-27.1693635	0.154958347
1	9.917334204	-26.91109959	0.258263912
1.033333333	9.917334204	-26.65283567	0.258263912
1.066666667	9.917334204	-26.65283567	0
		Break Distance	5.393794952

Table A.14. Brake test 2 at 75 PWM.

t	x	y	$\|\overrightarrow{xy}\|$
0.066666667	6.475890558	-20.28108361	
0.1	6.52609126	-21.48590046	1.205862243
0.133333333	6.52609126	-22.79111871	1.305218252
0.166666667	6.52609126	-23.79513275	1.00401404
0.2	6.576291962	-23.89553415	0.112252182
0.233333333	6.52609126	-24.9999496	1.10555578
0.266666667	6.576291962	-25.75296013	0.754682032
0.3	6.576291962	-26.10436504	0.351404914
0.333333333	6.626492664	-26.00396364	0.112252182
0.366666667	6.626492664	-26.05416434	0.050200702
0.4	6.576291962	-25.80316083	0.255974359
0.433333333	6.52609126	-25.60235802	0.206982797
0.466666667	6.626492664	-25.45175592	0.181001205
0.5	6.726894068	-25.35135451	0.141989027
0.533333333	6.576291962	-25.30115381	0.158748558
0.566666667	6.576291962	-25.40155521	0.100401404
		Break Distance	3.531445143

Table A.15. Brake test 3 at 75 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
0.566666667	17.62949163	34.98608092	
0.6	17.62949163	36.24143171	1.255350797
0.633333333	17.57491116	37.3330411	1.092973049
0.666666667	17.62949163	38.09716768	0.766073395
0.7	17.62949163	38.20632861	0.109160939
0.733333333	17.62949163	38.42465049	0.218321878
0.766666667	17.62949163	38.37007002	0.054580469
0.8	17.57491116	38.09716768	0.278306879
0.833333333	17.62949163	37.8788458	0.225041041
0.866666667	17.62949163	37.8788458	0
0.9	17.62949163	37.71510439	0.163741408
0.933333333	17.62949163	37.49678251	0.218321878
0.966666667	17.62949163	37.38762157	0.109160939
		Break Distance	3.235681874

Table A.16. Brake test 4 at 75 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
1.033333333	11.27887121	18.87041914	
1.066666667	11.27887121	19.41267257	0.542253424
1.1	11.17042053	19.30422188	0.153372429
1.133333333	11.17042053	19.41267257	0.108450685
1.166666667	11.17042053	19.84647531	0.433802739
1.2	11.06196984	19.95492599	0.153372429
1.233333333	11.06196984	19.95492599	0
1.266666667	11.17042053	19.95492599	0.108450685
		Break Distance	0.804076538

Table A.17. Brake test 1 at 65 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
0.233333333	15.43655708	17.09975515	
0.266666667	15.48853202	17.72345442	0.625861152
0.3	15.43655708	18.3471537	0.625861152
0.333333333	15.38458214	18.6070284	0.265021231
0.366666667	15.38458214	18.55505346	0.05197494
0.4	15.48853202	18.55505346	0.103949879
0.433333333	15.54050696	18.3471537	0.214298166
0.466666667	15.54050696	18.03530406	0.311849638
0.5	15.54050696	17.8274043	0.207899759
0.533333333	15.5924819	17.8274043	0.05197494
		Break Distance	1.832829704

Table A.18. Brake test 2 at 65 PWM.

t	x	y	$\lvert\overrightarrow{xy}\rvert$
1.033333333	13.07028626	13.12215247	
1.066666667	13.12215247	13.17401869	0.073349905
1.1	13.07028626	13.84827949	0.676252711
1.133333333	13.07028626	14.15947678	0.311197292
1.166666667	13.17401869	14.26320921	0.14669981
1.2	13.07028626	14.15947678	0.14669981
1.233333333	13.12215247	14.15947678	0.051866215
1.266666667	13.12215247	13.9001457	0.259331077
1.3	13.07028626	13.53708219	0.366749526
1.333333333	13.17401869	13.38148355	0.187006299
		Break Distance	2.14580274

Table A.19. Brake test 3 at 65 PWM.

A.5 Velocity Test

This data is based on the prebrake Brake Distance Test A.4 data, which is not included in the report.

Test	Velocity, V
1	$\frac{60{,}153cm}{1{,}033s} \Rightarrow 58,22\frac{cm}{s} \Rightarrow 2,10\frac{km}{h}$
2	$\frac{7{,}675cm}{0{,}133s} \Rightarrow 57,56\frac{cm}{s} \Rightarrow 2,07\frac{km}{h}$
3	$\frac{15{,}194cm}{0{,}267s} \Rightarrow 56,98\frac{cm}{s} \Rightarrow 2,05\frac{km}{h}$
Average	$57,59\frac{cm}{s} \Rightarrow 2,07\frac{km}{h}$

Table A.20. Velocity at 100PWM

Test	Velocity, V
1	$\frac{46{,}616cm}{1{,}067s} \Rightarrow 43,70\frac{cm}{s} \Rightarrow 1,57\frac{km}{h}$
2	$\frac{10{,}643cm}{0{,}233s} \Rightarrow 45,61\frac{cm}{s} \Rightarrow 1,64\frac{km}{h}$
3	$\frac{23{,}300cm}{0{,}533s} \Rightarrow 43,69\frac{cm}{s} \Rightarrow 1,57\frac{km}{h}$
Average	$44,33\frac{cm}{s} \Rightarrow 1,59\frac{km}{h}$

Table A.21. Velocity at 85PWM

Test	Velocity, V
1	$\frac{15{,}737cm}{0{,}533s} \Rightarrow 29,51\frac{cm}{s} \Rightarrow 1,06\frac{km}{h}$
2	$\frac{21{,}66cm}{0{,}7s} \Rightarrow 30,943\frac{cm}{s} \Rightarrow 1,11\frac{km}{h}$
3	$\frac{16{,}063cm}{0{,}567s} \Rightarrow 28,34\frac{cm}{s} \Rightarrow 1,02\frac{km}{h}$
Average	$29,60\frac{cm}{s} \Rightarrow 1,06\frac{km}{h}$

Table A.22. Velocity at 75PWM

Test	Velocity, V
1	$\frac{17{,}342cm}{1{,}033s} \Rightarrow 16,78\frac{cm}{s} \Rightarrow 0,60\frac{km}{h}$
2	$\frac{3{,}509cm}{0{,}233s} \Rightarrow 15,04\frac{cm}{s} \Rightarrow 0,54\frac{km}{h}$
3	$\frac{17{,}432cm}{1{,}033s} \Rightarrow 16,87\frac{cm}{s} \Rightarrow 0,61\frac{km}{h}$
Average	$16,23\frac{cm}{s} \Rightarrow 0,58\frac{km}{h}$

Table A.23. Velocity at 65PWM

Stalker B

B.1 The Distance Task

```
1  int ourRound(float f)
2  {
3      if ((f - floor(f)) >= 0.5)
4      {
5          return (int)ceil(f);
6      }
7      else
8      {
9          return (int)floor(f);
10     }
11 }
```

Code snippet B.1. The ourRound() function.

OSEK Kernel C

C.1 U32 Systick_get_ms(void)

```
1  // Called at 1000Hz
2  void systick_isr_C(void)
3  {
4    U32 status;
5
6    /* Read status to confirm interrupt */
7    status = *AT91C_PITC_PIVR;
8    // Update with number of ticks since last time
9    systick_ms += (status & AT91C_SYSC_PICNT) >> 20;
10   // Trigger low priority task
11   *AT91C_AIC_ISCR = (1 << LOW_PRIORITY_IRQ);
12 }
13
14 U32 systick_get_ms(void)
15 {
16   // We're using a 32-bitter and can assume that we
17   // don't need to do any locking here.
18   return systick_ms;
19 }
```

Code snippet C.1. The systick_get_ms() implementation.