Institut for Datalogi
Aalborg Universitet
Selma Lagerlöfs Vej 300
9220 Aalborg Ø.
Phone: 9940 9940
Fax: 9940 9798
Email: i16@cs.aau.dk
http://cs.aau.dk

Title: InfraRed Interceptive System

Project period:
SW5, fall 2009

Project group:
S503A

Participants:
Anton Møller Bentzen
Frederik Højlund
Lasse Andreasen
Jens-Kristian Nielsen
Jan Sundgaard Schultz

Supervisor:
Thomas Bøgholm

Number of pages: 74

Attachments:
CD-ROM

Finished: 18-12-2009

Abstract:

The purpose of this project is to research theory concerning embedded real-time systems and build such a system in the form of a LEGO robot, called IRIS. The tasks in this real-time system are scheduled using the cyclic executive approach. The robot is built on the MINDSTORMS NXT platform flashed with the LeJOS NXJ JVM. The result is a robot that is capable of shooting a target moving in a straight line at steady velocity.

Anton Møller Bentzen

Frederik Højlund

Lasse Andreasen

Jens-Kristian Nielsen

Jan Sundgaard Schultz

Preface

This report is written as a fifth semester project with the theme "Embedded systems", by five Software Engineering students from Aalborg University, Denmark. The report is written over a period of four months during the fall and winter of 2009. A list of sources are grouped in the bibliography and references throughout the report point to this chapter. Figures and tables are numbered in accordance with their containing chapter: the first figure in Chapter 1 is numbered 1.1, the second 1.2 and so forth. Explanatory text is written underneath figures and tables. To fully understand this report it is expected that the reader has knowledge of computer topics similar to the level of a student that has finished the fourth semester of Computer Science. The product of this project is a LEGO robot and the software to control it. Pictures of the robot and the source code is included on the attached CD.

Contents

Chapter 1

Introduction

The development of an embedded real-time system in the form of a robot is created using LEGO MINDSTORMS NXT and the LeJOS NXJ JVM.

First the theory behind embedded real-time systems is covered and the NXT hardware and the JOP are examined. Different methods for scheduling are examined and one fitting the tasks of the robot is selected for implementation.

The capabilities of the sensors and actuators are explored through experimentation and the robot is designed based on the gathered information. The sensor used to track the target is called an IRSeeker (InfraRed Seeker). This sensor can determine the direction of infrared emitting sources.

This report includes the implementation of a scheduler and the software used to control the robot. In order to make the robot a real-time system a cyclic executive scheduler is implemented to control the execution of the software. Fixed-priority scheduling is another method of scheduling. The possibility of using this scheduler for IRIS is analysed and tested but not implemented. The final product of this project is an autonomous embedded real-time system that tracks a moving infrared emitting target and shoots it with a projectile. The robot is called IRIS (InfraRed Inceptive Sentry) and is a stationary sentry used to intercept targets that enters its range.

1.1 Initiating Problem

In real-time systems it is important to know the time it takes to execute a piece of code. It is important that these systems operates within certain timing constraints, for instance in a car accident the airbag must release in time. The code used to detect the accident and release the airbag must have a known execution time in order to guarantee the effect of the airbag. Deadlines are introduced in order to control that the pieces of code execute in time. To get a practical view of the development of real-time embedded systems an autonomous

sentry is designed and implemented using LEGO MINDSTORMS bricks, sensors etc.

This leads to the following initiating problem:

- How are embedded real-time systems developed?
- What methods can be used to assure that real-time deadlines are met?
- How can an autonomous sentry using LEGO MINDSTORMS be developed?

1.2 Approach

This section explains the structure of this report. The analysis chapter explains theory concerning real-time systems and embedded hardware as well as experiments with different sensors in order to discover their capabilities and limitations. The design chapter covers prototype testing and choices regarding the design of the software and hardware. The implementation chapter covers topics about the implementation of the design. The conclusion chapter sums up the project and lists what is achieved during the semester. The reflection chapter reflects on the future development and utilization of IRIS in the real world.

Chapter 2

Analysis

Developing a real-time system that can track and shoot at a moving target is a challenging process. In order to achieve this the following chapter will cover the general theory about real-time systems. The first part is focused on this theory, and the second part is focused on the hardware that the robot will be implemented on. The hardware is first thoroughly described, and then a series of experiments are conducted to obtain information about their capabilities and limitations.

2.1 Real-Time Systems

This section will give an in-depth explanation of real-time systems, including deadlines, reactive and time aware systems, timeouts, tasks, scheduling and Worst Case Execution Time, in short everything required in order to create a simple real-time system. This section is based on [16].

A Real-Time System (RTS) is a system that has real-time constraints, this means that it is dependent on deadlines. In short, a deadline is a point in time, where the system is required to finish a task. Embedded systems are often Real-Time Systems (RTS's), for instance airbags, satellites, washing machines and cellular phones. Today 99% of all microprocessors produced are used in embedded systems [11].

A real-time system is a system that is required to perform actions when it is affected by its environment including the passage of time. A real-time system is required to react within deadlines.

The next section explains how the different types of RTS's can then be divided into soft, firm or hard RTS's.

2.1.1 Soft, hard and firm deadlines

Soft real-time systems are systems, where deadlines can occasionally be missed without system failure. For instance, collecting data from an input sensor at a regular basis. In the case that a deadline is exceeded the collected data will be included in the final calculations, it will however be de-emphasised gradually, compared to other results from the sensor.

In hard RTS's all deadlines must be met, if not the system fails. A pacemakers is an example of this. If it gives a delayed signal it causes potentially fatal heart failures.

Firm real-time systems are systems in which a deadline is allowed to occasionally be missed, but the data from that task will be discarded. An example is video decoding, as pixilation can occur if frame data is not processed fast enough. This will not cause a failure, but delayed data is not used.

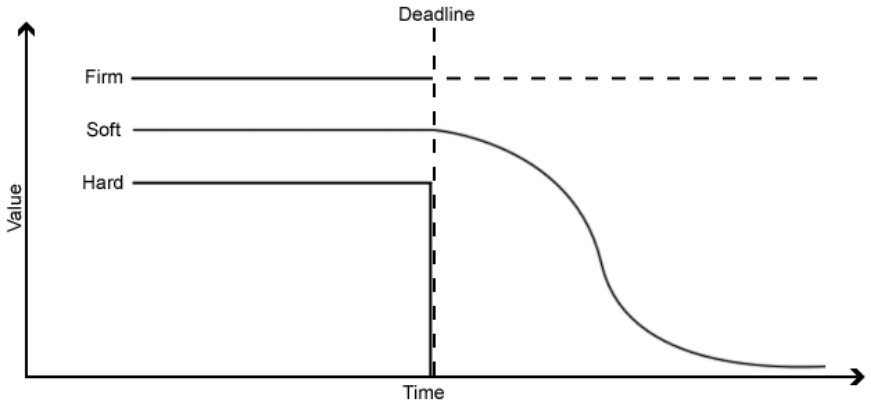

Figure 2.1: Depicting deadlines

Figure 2.1 shows how values are handled when deadlines are exceeded. The values of soft, firm and hard deadlines, before the deadline is reached, are the same. To maximise the readability on the figure the deadlines are separated.

In the soft system the delayed values will decrease in usefulness after the deadline has been exceeded. Values received after the deadline may still be somewhat useful but as more time passes values become less useful. After the deadline in a firm system is exceeded the value will not be used, and the system does not fail. Hard systems have no tolerance for deadline violations. If this occurs the result is a system failure.

2.1.2 Reactive and Time-aware Systems

A real-time system can be either reactive or time-aware. A reactive system works with relative times, for instance, an output must be produced 200 ms after any given input. Since a reactive real-time system is said to keep up with the environment, a reactive system must be synchronised with it. A time-aware system makes explicit references to the time of the

environment, said in other words these systems work with absolute times, for instance, the safe door of the bank must be closed at nine o'clock.

As mentioned a reactive real-time system can keep up with the environment, but only if it is made to be time-triggered. In a time-triggered system all computation activities happen at regular intervals, and an internal clock will release them for execution.

Real-time systems can also be made to be event-triggered, which means the environment will control the release for execution of software. The system has no control over when the different events are triggered.

2.1.3 Timeouts

Timeouts are used to detect "non-occurrences" of events, which is an important tool for error detection.

For instance a water boiler must have a mechanism for shutting off the boiler if its thermometer fails. If the thermometer breaks and the task that reads the temperature never returns a value, it is possible to use timeouts in order to detect this failure and shut off the boiler.

2.1.4 Tasks

A task in a real-time system is a piece of code that defines one of the subtasks of the system. Tasks have a number of properties and terms which are used in this report. They can be seen in table 2.1.

Description	**Notation**
Worst-case execution time - The longest possible execution time for a task.	C
Deadline - The point in time where tasks must be finished executing.	D
Interference time - The time a runnable task must wait on the execution of higher priority tasks.	I
Number of tasks in the system	N
Priority - Used in priority-based scheduling to decide order of execution.	P
Worst-case response time - The maximum time between release of a task and end of computation.	R
Period - The minimum time between task releases.	T
Utilization - The time spent on executing divided by the period of the task.	U

Table 2.1: Common terms and notations regarding tasks.

These terms will be elaborated as they are used throughout the report.

A task can have a total of five different states:

- Non-existing
- Created
- Initializing
- Executable
- Terminated

A simple example of a life-cycle of a task could be: A task i is created, the task then moves into initialization. If it fails to initialise, the task is moved into the state terminated without executing. If it succeeds it enters the executable state, where the task may wait for resources, e.g. the CPU. After execution the task proceeds into the state terminated, which leads to the state non-existing. In this state the task cannot be accessed.

The creation, dispatching of available resources and termination of tasks are undertaken by the Run-Time Support System (RTSS). RTSS can be implemented as either a hardware structure or a software application.

A life-cycle for a task can be expanded with two extra states:

- Waiting Child Initialization
- Waiting Dependant Termination

The states can be seen in figure 2.2, which shows the expanded version of the life-cycle of a task. The waiting child initialization state is a state, in which the task can placed, if it is responsible for the creation of other tasks, also called the parent task. This means that the task is delayed until the tasks within it, the children, are created and initialised. The task then moves on in the life-cycle. Waiting dependant termination is, when other tasks are dependant on the task or tasks within it. The task upon which they are dependant is called the guardian task. The guardian must not terminate before all dependant tasks have terminated.

When considering the interaction of tasks, there are three types of behaviour:

- Independant behaviour
- Cooperating behaviour
- Competing behaviour

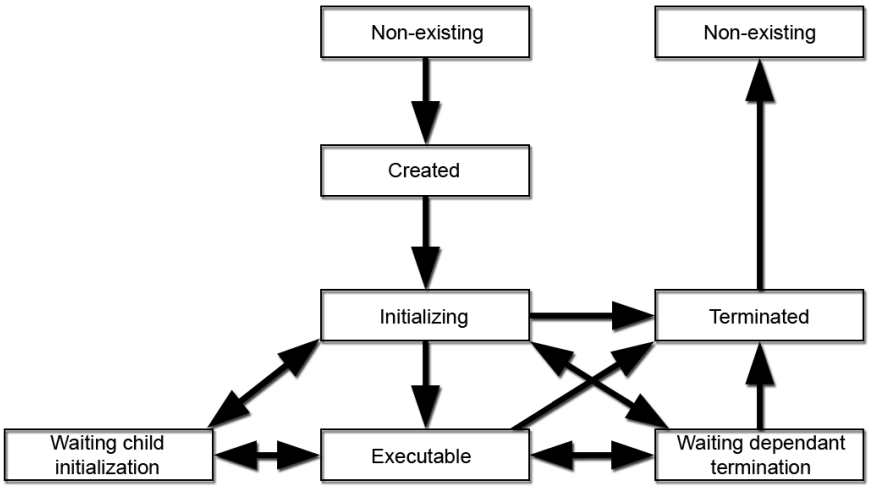

Figure 2.2: This is a state diagram for any task

Independant tasks never communicate, synchronise or exchange data. Cooperating tasks often communicate and synchronise in order to achieve common goal or perform common actions. An example of this could be a component in an embedded system that needs to involve several tasks in order to keep a temperature in a boiler within a range. It fails if the heating and measure tasks do not communicate. Tasks with competing behaviour will compete for system resources, it being peripheral devices, e.g. system memory, computation time etc., since those resources are limited. This will require tasks to synchronise and communicate to be able to allocate the resources in a needed way. Despite that tasks communicate, they remain independant when considering the purpose of the task at hand.

2.1.5 Scheduling

If the exact order in which tasks execute is not specified in a concurrent system, the behaviour of the system becomes non-deterministic. The program may perform despite non-determinism, however the temporal requirements may not be fulfilled, since the timing behaviour may vary considerably. For instance, a non-deterministic airbag system may release the airbag at some point, but not necessarily in time to have an effect. The purpose of scheduling is to eliminate this non-determinism and ensure that it is a real-time system. It is performed by using a scheduling algorithm, which provides the means of ordering tasks. Scheduling algorithms also make it possible to predict the worse-case behaviour of the system, which provides the means for determining whether or not the timing constraints are satisfied.

Scheduling a system can be done using two different methods: static and dynamic scheduling.

- In a static scheduling scheme, predictions on task-execution-order are made before

executing.

- In a dynamic scheduling scheme the execution-order will be decided during runtime.

The Cyclic Executive Approach

The cyclic executive approach is one of many different methods for scheduling a real-time system. The concurrency problem discussed above is handled by avoiding it. With this approach a fixed set of purely periodic tasks are placed together in minor cycles within a major cycle, such that repeated execution of the major cycle will ensure that all tasks are run at their correct rate. The cyclic executive will be elaborated in section 2.2.

2.1.6 Priority-based scheduling

When a system becomes more complicated, the cyclic executive approach becomes less appealing, since it is more appropriate for dealing with simple systems with few tasks. An alternative to the cyclic executive approach is priority-based scheduling.

Priority-based scheduling, sometimes referred to as task-based scheduling, introduces the concept of tasks to be run concurrently. During runtime a scheduler will determine which task is to be run based on the priority of a task. There are different methods to calculate and assigning the priorities, some of those will be explained shortly.

FPS

Fixed Priority Scheduling (FPS) is the most widely used form of scheduling, using FPS the tasks are assigned a static fixed priority. FPS is elaborated in section 2.3.

EDF

Earliest Deadline First (EDF) scheduling is another approach, where the tasks are assigned priorities according to their absolute deadlines. This means that the task with the earliest deadline will have the highest priority. An absolute deadline is a deadline at a certain point in time, e.g. at 5 pm, where a relative deadline is some point in time relative to the current time. In EDF the absolute deadlines must be calculated using the relative deadlines of each task. Because EDF can change the priorities of tasks during runtime, EDF is dynamic. EDF is more complex and difficult to implement than FPS given that EDF has dynamic task priorities. Consequently EDF will not be discussed in further details.

VBS

VBS, or Value-Based Scheduling is used when there is a risk of overloading the system. If the system becomes overloaded, meaning that the utilization is greater than 100% , the normal use of static priorities or deadlines is not sufficient. Instead, one can assign values to each task and employ an online value-based scheduling algorithm. Because VBS is a rarely used approach for scheduling, it is therefore not relevant to this project and will not be elaborated.

Preemption and non-preemption

It is not possible to release a task during execution of another task in cyclic executive scheduling, it is however an option in priority-based scheduling. If a high-priority task is released during execution of a low-priority task, the release can be handled in two ways:

- In a preemptive scheme the system will interrupt the low-priority task and immediately switch to the high-priority task.
- In a non-preemptive scheme the low-priority task will continue to execute, and when it is finished, the system will switch to the high-priority task.

If non-preemptive systems must be able to switch between tasks, cooperative multitasking must be implemented. Cooperative multitasking is when the task currently using the CPU must sacrifice the control for another task. Non-preemptive systems are not used as often as preemptive systems. This is because a non-preemptive system will not allow a high-priority task to interrupt a low priority task, which means that aperiodic and sporadic tasks are not as well supported. In a non-preemptive system the task will not be interrupted by a task with a higher priority. An example of this could be, when reading data from one sensor the task will not be interrupted by the release of a task with a higher priority, meaning that reading from the sensor will not be interrupted.

2.1.7 Worst-case Execution Time

Worst-case Execution Time (WCET) is the maximum amount of time a task execution can take. WCET analysis can be done using either analysis or measurement. Both methods have their issues. Analysis is usually the most precise, however an effective analysis-model for the processor in use must be available. The model must include every parameter including caches, pipelines, branch prediction, out-of-order execution, memory wait states etc. With the measurement method it can difficult to ensure when the worst case has been observed, which means that often, one must make unnecessary overhead.

A typical analysis technique consists decomposing code of a task into machine code for the given platform. Then the machine code is given as input to the WCET-analysis-model of the platform. Analysing loops to determine the number of iterations is an undecidable problem, so loops are collapsed using knowledge about maximum bounds. Simple choice-constructs are also collapsed to single values. If sufficient information is available more sophisticated reduction techniques may be used.

The Java Optimized Processor, which will be described in section 2.6 is specifically designed for proper WCET-analysis. This has made it possible to use tools that automatically calculates WCET for a task, when passing the code of the task as input [12]. To utilise WCET-analysis the code must be confined in certain ways. Predicting the time a loop requires to finish is difficult, hence it is necessary to determine a static value for the maximum loop bounds. Information about maximum loop bounds must be explicitly passed to the tool, for instance with specially formatted comments.

Modern multi-core-processors with on-chip caches, pipelines, branch prediction etc. are often made to reduce average time execution time, but these features affect WCET prediction in a negative way. Ignoring such things often results in very pessimistic models, but not doing so can make the model extremely complex.

2.1.8 Summary

Real-time systems require a range of constraints to be satisfied in order to function properly. In the discussion of real-time systems, the following terms were introduced:

- **Tasks** - A task in a real-time system is a piece of code that defines one of the subtasks of the system.
- **Deadlines** - A point in time, by which a task must be completed, or else the RTS will fail depending on if it is a soft, hard or firm RTS.
  - **Soft** - A task may exceed its deadline, the usefulness of the output from the given task is decreased over time, moving toward zero.
  - **Hard** - A system that makes use of hard deadlines will fail if a deadline is exceeded.
  - **Firm** - A task may exceed its deadline, if this occurs the output from the task will simply be discarded.
- **Reactive and Time-aware** - Reactive systems are synchronised with the environment, meaning they function with relative time. Time-aware systems make use of static timestamps, e.g. 9 PM every evening.

- **Timeouts** - Timeouts are introduced as a method for detecting when deadlines are exceeded.
- **Preemption and non-preemption** - Releasing a task during execution may be done in two ways, preemption interrupts the low priority task and switches to the high priority. Non-preemption will switch to the high priority task once the low priority task has finished executing.

With the basics on what a task is, are covered, the various methods for scheduling these tasks into a RTS are discussed.

- **Cyclic Executive Approach** - A major cycle, consisting of minor cycles, is created. If this can be done, the system has been proven to be cyclic executive schedulable.
- **FPS** - Fixed Priority Scheduling gives each task a static priority prior to execution.
- **EDF** - Earliest Deadline First gives each task a priority corresponding to how early the deadline appears, the first deadline is given the highest priority.
- **VBS** - Value-Based Scheduling is used when there is a risk of overloading the system.

These methods each result in a scheduled system being output, these may in turn, apart from the cyclic executive schedule, be analysed for a Worst-Case Execution Time.

- **WCET** - via analysis or measurement the WCET of a system may be determined.

With the basics on RTS's established a more thorough explanation of the Cyclic Executive Approach and the Fixed Priority Scheduling algorithm is given. The cyclic executive approach is a simple way of scheduling a system and if this is applicable, so is the FPS algorithm. Fixed-priority Scheduling (FPS) being the algorithm used in the majority of RTS's makes it a very important discussion.

2.2 The Cyclic Executive Approach

As described in the previous sections, there are different methods for scheduling a RTS. In the following section the cyclic executive approach will be elaborated.

The cyclic executive approach is a way to schedule a fixed set of tasks, each with a fixed period T and a worst-case execution time C. When using the cyclic executive approach, tasks are divided into small segments of code which are assembled into a table. The segments of code that are stored in this table are then ordered in such a way that the system complies with all deadlines, periods and other timing constraints. If it is possible to compose a single

"completed table" or "major cycle" from the set of tasks in question, then the set of tasks is schedulable with the cyclic executive approach. The major cycle consists of a number of minor cycles of fixed durations e.g. having four minor cycles each with a duration of 25 ms the resulting major cycle will have a duration of 100 ms. During execution an interrupt will occur every 25 ms. At the occurrence of these interrupts, the minor cycle will begin to execute. The schedule will order these tasks in a way that guarantees each task will run at its proper rate, because each minor cycle is assured to finished before the next cycle begins.

Example

The first part is the table, which will describe the periods and the computation times of each task, see table 2.2, for the tasks in this example. It is seen that each task has an execution time that fits within its period. Figure 2.3 shows a possible construction of the cyclic executive. The minor cycles are constructed in a way such that each task is executed at its correct rate. After the fourth minor cycle, all the cycles can be repeated.

Task	**Period, T**	**Computation Time, C**
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

Table 2.2: Cyclic executive task set

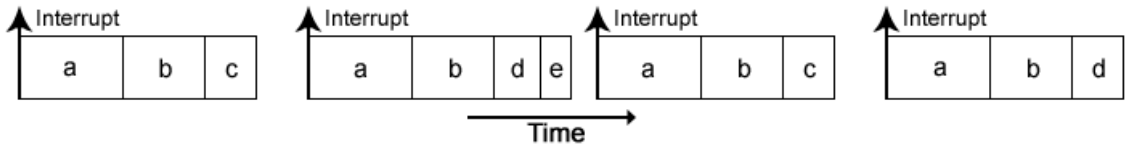

Figure 2.3: Time-line for the task set

If the cyclic executive can be constructed, it is not necessary to do any further scheduling. The cyclic executive approach does however have some issues. It is known to be a problem of NP-hard complexity to create a cyclic executive, similar to the bin-packing problem. It is computationally infeasible to make a cyclic executive with a large set of tasks. Another problem with the cyclic executive is the trouble of including tasks with either a long period or long execution time. Sporadic tasks also makes it difficult to construct cyclic executive, since they are difficult to predict. It is a simple method and if it is possible to schedule the tasks this way, it is definitely worth considering this method.

2.2.1 Summary

The cyclic executive is mostly used for small systems that does not contain many tasks. In this section the basics of the method were covered and the following terms were introduced:

- **Minor Cycle** - A small segment of code that must be run
- **Major cycle** - A collection of all the minor cycles within a schedule

By breaking the program into small pieces of code, creating minor cycles and assembling these into major cycles, the system can be scheduled. It is important to note that proving a system may be scheduled using the cyclic executive approach is done by using so-called "proof by construction". This means that if the major cycle can be constructed, it can be repeated, and the system is therefore schedulable.

Following this section is an explanation of another very different scheduling method. The FPS method is used in a majority of RTS's, and it is an important tool in the real-time world.

2.3 Fixed-priority Scheduling

The following section explains a thread-based approach to scheduling called fixed-priority scheduling. Fixed-priority scheduling allows the tasks to be assigned static priorities prior to execution.

FPS is a widely used approach for scheduling. Before the scheduler is executed, each task receives a fixed static priority. These tasks are then executed in order of their priorities. Priorities are expressed by integers - the higher the integer, the higher the priority, for example 2 is a higher priority than 1. The priority of a task has nothing to do with its importance, but is derived from its timing constraints.

After priorities have been assigned, there are several ways to test whether or not the system can be scheduled using the given priorities.

2.3.1 Rate Monotonic Priority Assignment

One particular way of distributing priorities amongst tasks is using rate monotonic priority assignment. If the following conditions are met rate monotonic priority assignment can be used:

- The application is assumed to consist of a fixed set of tasks.

- The tasks are completely independent of each other, this means that there can be no blocking.
- The system is considered as being perfect, meaning that all switching between tasks occur instantaneously.
- All tasks have fixed WCET values.
- All tasks are executed on a single processor with a single core.

Using rate monotonic priority assignment, each task is assigned a unique priority based on the length of its period: The longer the period, the lower the priority.

$T_i > T_j \Rightarrow P_j > P_i$

An example can be seen in table 2.3. Since the task c has the longest period, it has been assigned the lowest priority. Remember that 1 is a lower priority than 2.

Task	**Period, T**	**Priority, P**
a	15	3
b	20	2
c	35	1

Table 2.3: An example of rate monotonic priority assignment

It is important to note that any task set that can be scheduled using preemptive priority-based scheduling with a fixed-priority assignment scheme, can also be scheduled using rate monotonic priority assignment.

2.3.2 Utilization-based Schedulability Testing

When using rate monotonic priority assignment a simple test can be performed to check for schedulability of a set of tasks The utilization-based test is sufficient, but not necessary, meaning that if this test passes, the task set is possible to schedule, however if the test fails it may still be schedulable. The formula (2.1) is the formula for a utilization-based test:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \tag{2.1}$$

In formula (2.1) C is the WCET of the task, T is the period, and N is the number of tasks in the set. Notice that the right-hand side of the equation only depends on N, as N goes towards infinitive the value of the right-hand side will decrease towards 69.3%. This means that any task set with a combined utilization of less than 69.3% will be schedulable

using rate monotonic priority assignment. In table 2.4 different variations of the utilization bound for N can be seen.

N	**Utilization bound**
1	100%
2	82.2%
3	78.0%
4	75.7%
10	71.8%
100	69.6%
1000	69.3%

Table 2.4: Utilization bounds.

2.3.3 Response Time Analysis

Response time analysis (RTA) is different from utilization-based testing, but it is used with the same purpose in mind: to test for schedulability. RTA is both sufficient and necessary, unlike utilization-based schedulability testing, but it is also more complex. Each task is analysed according to the priorities, computation times and periods. Using this information the worst-case response time for each task is calculated with the formula (2.2).

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{2.2}$$

The task with the highest priority will have a worst-case response time, denoted by R, equal to its own computation time: $R = C$, because no other task may run before. Any task with a lower priority will suffer from interference, which means that a low-priority task that is runnable must wait for the execution of a higher-priority task. Based on this, interference must be taken into consideration when calculating the worst-case response time for each task. The formula (2.3) shows the worst-case response time.

$$R_i = C_i + I_i \tag{2.3}$$

In formula (2.3) I_i is the maximum interference a task i may be postponed in the time interval (t,t+ R_i). Considering the scenario that all tasks are released at the same time, e.g. time 0, the lowest priority task will have to wait for all the other tasks to finish, hence this low priority task will suffer from the maximum interference possible.

Consider the task j with a higher priority than task i, the number of releases j has before

i is executed within the interval $[0, R_i)$ is calculated by the formula (2.4).

$$Number\ of\ Releases = \left\lceil \frac{R_i}{T_j} \right\rceil \tag{2.4}$$

The ceiling function $\lceil x \rceil$ will give the smallest integer that is greater than x, so if R_i is 15 and T_i is 6, it means that there are three releases of task j, since $\lceil \frac{15}{6} \rceil = 3$. Each release of j will add time equivalent to C_j upon the interference of i. This can be expressed as formula (2.5).

$$Maximum\ Interference = \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.5}$$

In order to acquire the total interference of task i, one must perform this calculation on all higher-priority tasks:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.6}$$

In formula (2.3) *hp(i)* is the set of all higher-priority tasks. By combining the equations (2.6) and (2.3) the equation (2.7) is derived.

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.7}$$

As it can be seen in the formula (2.7) R_i is represented on both sides, which makes it a fixed-point equation. In order to solve the equation (2.7) it must be converted into a recursive function, as formula (2.8) shows.

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

If $w_i^{n+1} = w_i^n$, the solution for worst-case response time for task i is w_i^n.

If, for all tasks in the system, the worst-case response time is lower than or equal to the deadline of the current task, the system will be schedulable. If the worst-case response time increases above the period of the current task, the task will exceed its deadline. Thus it can be concluded that the system cannot be scheduled.

2.3.4 Summary

Using FPS allows the system to provide tasks with static priorities, it also allows access to various tools for calculating WCET which is necessary in some systems. Knowing different scheduling methods will undoubtedly prove itself useful when been given that task of selecting the method best suited for the system one wishes to create.

Besides the general theory about RTS's, it is important to be familiar with the available hardware, which will be elaborated in the following part of the analysis.

2.4 Hardware in Embedded Systems

There are many things to consider when working with hardware and sensors in particular. In this section it will be explained what parameters should be taken into consideration when using sensors and actuators. This section is based on [9].

2.4.1 Actuators

An actuator can be described as a device that performs an action. Typical examples of actuators are speakers, small electric motors, the display on a mobile phone etc. Things like applied load to electric motors must be taken into consideration. If a motor does not perform as expected this will have to be handled by the system.

2.4.2 Sensors

A sensor is a device which monitors conditions in the real world. In the ideal world, a sensor would not alter the parameter it is measuring. This is, however, never the case, take for example a sound detector. In order for it to work it must absorb some of the sound waves, thereby changing the environment. Sensors have a number of attributes that should be taken into consideration when being used. Some of these also account for actuators:

- **Sensitivity**, how the output reflects the current input.
- **Efficiency**, the amount of energy required to run the sensor.
- **Resolution**, the accuracy of the device.
- **Response time**, the time from when the device receives some input until it is forwarded to its controller.
- **Stability**, the stability of the device.
- **Offset**, what the output from the device is when it receives no input.

2.4.3 Processor technology

There are different types of processor technologies: General Purpose Processor (GPP), Application-Specific Processor (ASP), and Single-Purpose Processor (SPP). In the world

of PC's most people know of Intel, AMD or even IBM CPU's. These are GPP's, as they can be programmed to do just about any calculation. GPP's require more energy to run compared to the other technologies. They may also suffer from a loss of efficiency, since the GPP is capable of performing multiple tasks. An example of a ASP is the Java Optimized Processor (JOP). This processor is built to execute Java code and has been optimised for this. Lastly, the SPP, the most used processor technology, is the technology that can be found in a lot of appliances. These processors are created with the goal of performing one single task and nothing else.

For example SPP's are used in everything from watches to electronic toys and gadgets. It can only perform one task, but it does so with great efficiency and low power consumption.

2.4.4 Post Processing

Post processing can be used to process data received from a sensor. Input from sensors can contain noise or other unwanted information and it is required to be filtered before it is used in the system. This can be done with algorithms written in software or it can be filtered in the hardware. The principle is the same, taking an input and creating an output that can be used for the current application. An example is a microphone that records analogue sounds. The signal needs to be post processed into a digital signal in order to be recorded onto a computer.

2.4.5 Summary

In this section different types of hardware and processor technologies were discussed, The following terms were introduced:

- **Actuators** - Is a device that performs a certain action. For example a display, a servo motor, a speaker or an electromagnet.
- **Sensors** - A device that senses a condition of the real world. For example a microphone, an infrared receiver or even a switch that can be turned on and off.
- **GPP** - General Purpose Processor, GPP's are often to be found in modern desktop computers.
- **ASP** - Application Specific Processor, ASP's appear in the form of a processor meant for a certain application, such as executing Java code.
- **SPP** - Single Purpose Processor, SPP's are the most common form of processors, these are in everything from watches, TV's, toys and remote controls, it is a processor meant for doing one task with great efficiency.

- **Post Processing** - Post processing is important when dealing with data from sensors. For example post processing can be converting the analogue signal from a microphone into a digital signal that may be recorded onto a CD.

There are many things to consider when working with sensors and actuators. The importance of each parameter may however vary from system to system. Some systems require high stability while others require high resolution. When creating a system with sensors and actuators one must carefully weigh the importance of each parameter. Following this brief explanation of hardware, the actual hardware available to this project will be described; including the NXT, available sensors and a short explanation of the available software.

2.5 LEGO MINDSTORMS NXT

This section will describe the NXT Brick and show a selection of the different NXT sensors. How they work and how the NXT Brick may be programmed to take advantage of them will be explained.

2.5.1 NXT Brick

Figure 2.4: The NXT Brick

This is a programmable computer, and it can be used to control output and input devices such as LEGO NXT sensors and actuators. A picture of the NXT Brick can be seen in figure 2.4.

An inside look reveals two processors as seen in figure 2.5. There is a main processor and a secondary processor which are described in table 2.5.

The NXT Brick is equipped with four input ports for connecting sensors and three output ports for connecting servo motors. The NXT Brick has a 100x64 pixel LCD display, and it

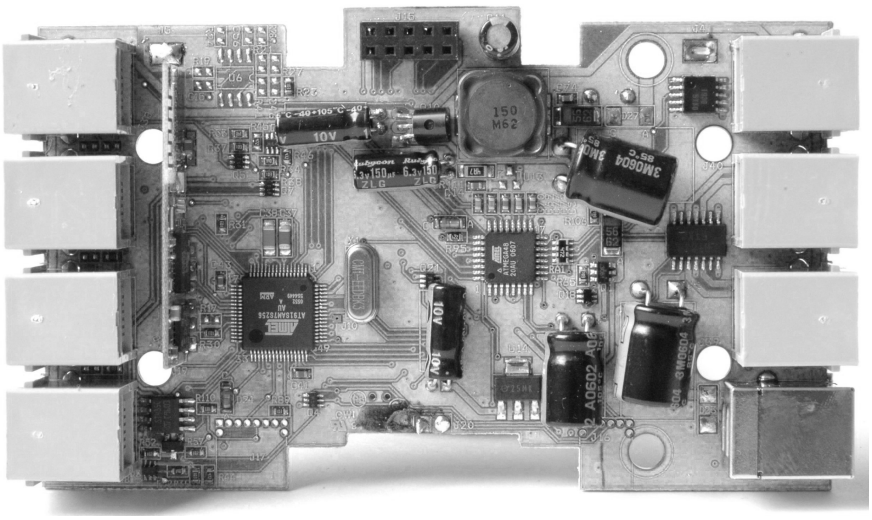

Figure 2.5: The circuit board of the NXT

Processor	**RAM**
48MHz 32-bit ARM7 processor	256KB flash, 64KB
8MHz 8-bit AVR Processor	4KB flash, 512B

Table 2.5: Description of the processors inside the NXT Brick

also features four programmable buttons positioned on top of the NXT Brick.
The NXT Brick can be connected to a computer using USB or Bluetooth, and via these connections programs can be uploaded to the NXT Brick.

2.5.2 NXT Sensors

Following is a descriptions of the sensor and actuators made available for this project. The selected sensors are chosen because they have some relevance to this project.

NXT Touch Sensor

The NXT touch sensor has a few simple functions: its button can be pushed, released and bumped. The sensor comes bundled with the NXT base kit and it connects to the NXT via one of the sensor ports. In addition, the orange and the three grey buttons on the NXT brick can be programmed and used in much the same way as the touch sensor [5].

Ultrasonic Sensor

The Ultrasonic Sensor works by emitting ultrasonic waves and measuring the time for these to return to the sensor. It calculates the distance to the object in front of it based on the time it took the waves to return.

According to its manual the ultrasonic sensor can measure distances from 1-255 centimetres[10].

Servo Motor

The servo motor has a built-in tachometer. This means that the angle or rotation can be precisely controlled using the NXT Brick. Using the tachometer it is possible to rotate the motor a certain number of degrees, control the speed of its rotation and measure the distance driven[6].

Infrared Distance Sensor

The High Precision Medium Range Infrared distance sensor for NXT (DIST-Nx-Medium-v2) is a third party sensor produced and sold by a company called Mindsensors.com. It is an optical distance sensor that outputs the distance to an object. According to its manual it can measure in the range of 100 mm to 800 mm. Mindsensors.com has provided various examples of code, allowing the distance sensor to work with various NXT operating systems, such as LEGOs own NXT-G, RobotC and LeJOS NXJ[8].

IRBall

The IRBall is an infrared emitting ball, it contains 20 infrared LEDs placed such that the ball emits infrared light in all directions. It also has four different modes, a pulse modulated and an un-modulated mode, a 1200 Hz and a 600 Hz mode. The advantages of the pulse modulated mode are that it makes it easier to detect in difficult lightning conditions, and it saves battery compared to other modes [3].

IRSeeker

The HiTechnic Infrared Seeker can detect infrared radiation. Inside its casing there is five infrared receivers each positioned at 60°intervals. According to its manual the area at which the sensor can detect is mainly in front of and to the side of the sensor, which can be seen in figure 2.6.

The IRSeeker will measure the infrared signal sent from the IRBall on all sensors and determine the direction of the IRBall based on those readings. The IRSeeker sensor provides the direction as a value from 1 to 9 based on which area the IRBall is located in. It is also possible to obtain the individual reading from each infrared receiver.

The IRSeeker can be set to operate in the two following modes [4]:

- **Modulated Mode (AC)** This mode will detect modulated infrared signals. The IRBall can be set to send modulated signals and this will allow the IRSeeker to more

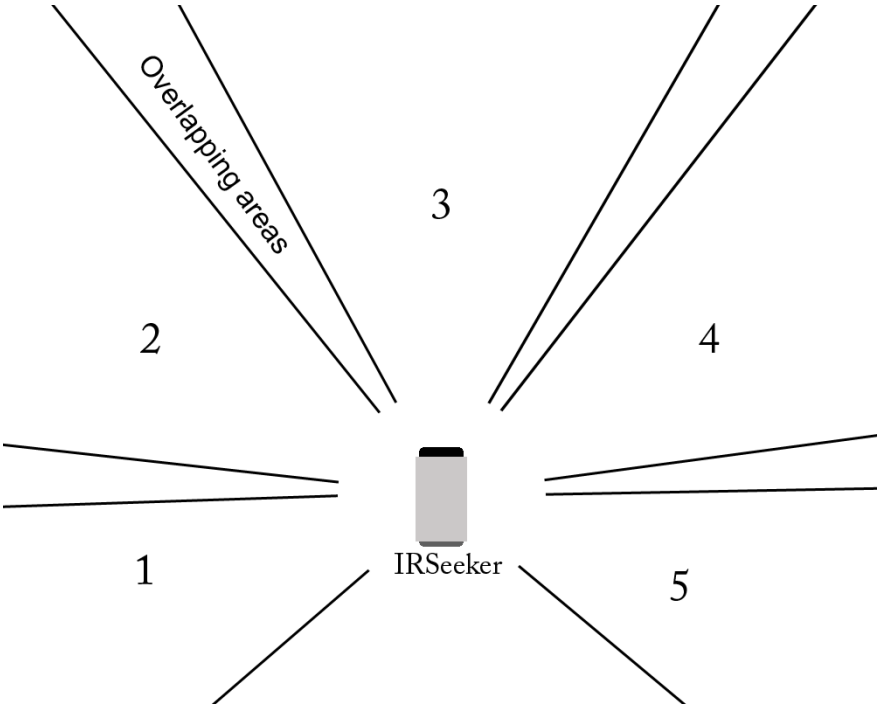

Figure 2.6: This shows the different areas of the IRSeeker

efficiently obtain the direction of the IRBall because it can filter out most other infrared light sources, such as sunlight.

- **Un-modulated Mode (DC)** This setting is more prone to be affected by sunlight, it is still available because older IRBalls cannot send a modulated signal.

2.5.3 Software

The NXT Brick comes with a standard firmware stored in the flash memory that can execute programs created with LEGO's NXT-G software. It is, however, possible to write programs in other languages and compile them to NXT-G programs [2]. There is actually quite a few of them and most are based on existing programming languages, for example Not eXactly C (NXC) is a language for the NXT with a syntax very similar to C.

Another possibility is to flash the NXT Brick with another firmware in order to use some languages. There are several different firmwares available, some of which include[2]:

- **LeJOS NXJ** can be used to program Java onto the NXT
- **pbLua** can be used to program LUA onto the NXT
- **LeJOS OSEK** can be used to program ANSI C, it is also required if you wish to program Ada onto the NXT

The LeJOS NXJ introduces a subset of Java to the NXT platform, which will be explained in the next section.

2.5.4 LeJOS NXJ

LeJOS NXJ is a Java based firmware for the LEGO NXT Brick, which means that the programs are written in Java. LeJOS supports most of the standard Java API. Some of the most significant features of LeJOS NXJ are [7]:

- Object oriented language (Java)
- Preemptive threads (tasks)
- Arrays, including multi-dimensional
- Recursion
- Synchronization
- Exceptions
- Java types including float, long, and String
- Most of the java.lang, java.util and java.io classes
- A well-documented Robotics API

Writing Programs With LeJOS NXJ

Below is an example of how to program with the LeJOS NXJ API. Anyone who has seen object-oriented code before should be able to see what the code does.

```
1 import lejos.nxt.*;
2
3 public class program {
4   public static void main(String[] args) {
5     Motor.A.setSpeed(50);
6     Motor.A.forward();
7     while (!Button.ESCAPE.isPressed()) {} // Busy-wait
8   }
9 }
```

For the sake of clarity the sample program is explained step by step. First the speed of motor A is set to 50 degrees per second. After this it is instructed to move forward[1]. It then busy-waits until the escape button on the NXT is pressed and exits the program.

[1]Forward is a fairly ambiguous term in this context as LeJOS NXJ have no sense of direction other than forward always is one direction and backwards the other.

The LeJOS API has a fair amount of classes ready for use to get data from sensors and control motors. It requires little to none new low level code to get a robot running, as most, if not all the standard LEGO sensors are supported directly in the API. In addition a fair amount of third-party sensors from companies like Mindsensors.com and HiTechnic are also supported. If a particular sensor is not supported in the API, it is simple to access the raw data from the sensor via the I2C bus.

2.5.5 Summary

The preceding section describes some of the sensors and some of the software available to control these sensors. The LeJOS NXJ API is a Java based API, it contains a subset of the standard Java library and is capable of doing complex calculations. There are platforms other than the NXT Brick that makes use of the Java programming languages, LEGO sensors and actuators, one of these being the Java Optimized Processor which will be described in the next section.

2.6 Java Optimized Processor

The JOP is a hardware implementation of the Java Virtual Machine, created as an alternative to a layered architecture. This means that Java bytecode can be translated directly to the microcode of the JOP, saving valuable resources. This is usually important for embedded systems, where resources are often sparse. The JOP is developed such that it has very predictable execution time of instructions. It was developed as a part of Martin Schoeberl's Ph.D. thesis at the Vienna University of Technology, Austria. The JOP is small and can easily be implemented in low cost field programmable gate arrays and used in various situations. The JOP is now an open-source project under the GNU General Public License, version 3 [14].

The specifications of the JOP can be viewed in table 2.6.

Target technology	Altera, Xilinx FPGA
Memory	3 KB
Clock	100 MHz

Table 2.6: JOP specifications[13].

2.6.1 Time Predictability of the JOP

When designing real-time systems, static WCET analysis is necessary and a processor model, which is designed for easy calculation of WCET is needed. Traditionally only simple pro-

cessors can be analysed properly for WCET. Modern processors with architectural advances are being designed to have fast average-case execution time rather than a fast WCET, which effects WCET analysis negatively. The architecture of the JOP is designed to be simpler and more accurate for WCET analysis.

The JOP is a RISC stack processor with a simple 3-stage pipeline with an additional stage that fetches bytecode instructions and translates it into microcode. The JOP makes it possible to make accurate WCET analysis by performing Path Analysis at the microcode level. Path Analysis generates a control flow graph (a directed graph of basic blocks) of the program and annotates (manual or automatic) loops with bounds. This is possible because each microinstruction takes a specific time to execute[15].

2.6.2 LRBJOP

The JOP can be used to control LEGO MINDSTORMS sensors and actuators instead of LEGO's own microcontroller. This can be done with a printed circuit board called LRBJOP, which is also an open-source project. The LRBJOP lets the JOP interface with MINDSTORMS RCX sensors and actuators, but not the NXT gear. There exists an easy-to-use Java package that allows access to the LEGO RCX sensors. The RCX is the old version of LEGO MINDSTORMS computer. There are not as many RCX sensors compared to NXT sensors and they lack in accuracy.

This package enables the programmer to abstract from implementation details and allows intuitive use of the sensors and actuators[1].

2.6.3 Summary

This section introduced the JOP which has easy WCET analysis. The main disadvantage of the JOP seen in the light of this project is its incompatibility with the NXT sensors and actuators.

The next section will describe some in-depth and well documented experiments performed with the NXT and its sensors.

2.7 Experiments

In order to evaluate and give arguments for what sensors to use and why, a number of experiments were carried out. These tests examined the accuracy of the available sensors and the limitations to expect.

2.7.1 Ultrasonic sensor experiment

Goal

The goal of this experiment is to determine the environment in which the ultrasonic sensor can operate and the accuracy of the measurements it makes.
The IRBall was mounted on a LEGO wagon and moved in front of the sensor. This distance will be measured and compared to the real distance.

Description

A program was written for this experiment that would read the data from the ultrasonic sensor every 500 ms and output it on the display of the NXT Brick. The distance from the sensor to the IRBall wagon was initially 0 cm, and the distance was then increased by 5 cm for each measurement.

Setup

Figure 2.7 shows the setup used for the experiment. The sensor was stationary and the IRBall wagon was then to be moved relative to the sensor. It is important to note that the IRBall wagon was elevated to the same height as the sensor and was not moved vertically.

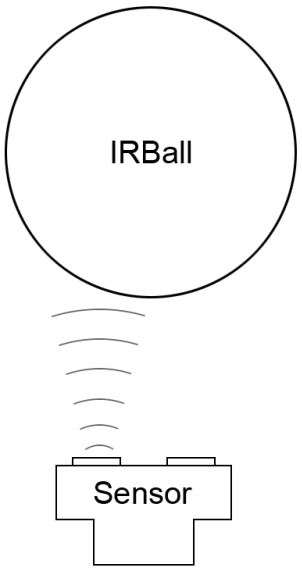

Figure 2.7: The test setup of the ultrasonic sensor experiment

Result

The results from the experiment are listed in table 2.7.1, The table shows the actual measurements (cm) along with the sensor readings (cm). Secondly, figure 2.8, shows a graph

depicting the deviation in percentages.

Actual (cm)	**Sensor (cm)**
0	6
5	7
10	15
15	22
20	23
25	27
30	31
35	36
40	40
45	46
50	52
55	58
60	64
65	68
70	74
75	83
80	85
85	90
90	97
95	100
100	104

Table 2.7: Readings in cm from the experiment with the Ultrasonic Distance Sensor

Sources of errors

- The sensor is dependent of the reflection on the sound waves it emits.
- Different shapes and materials can affect the sound waves.

Conclusion

The experiment shows that the sensor was relatively precise when measuring the wagon with the IRBall, especially from about 20 to 70 cm, as seen in the graph 2.8. During the experiment it was discovered that the sensor was sensitive to certain shapes, for instance a box with one of the corners turned towards the sensor would be completely invisible, as the sound waves were reflected away from the ultrasonic sensor. This effect is illustrated in figure 2.9. The experiment also shows that the ultrasonic sensor will measure in a 23-24 cm wide beam up to a distance of one meter.

There are several problems to consider when using the ultrasonic sensor. It can only target in one direction, and in order to keep track of a moving object the sensor must be

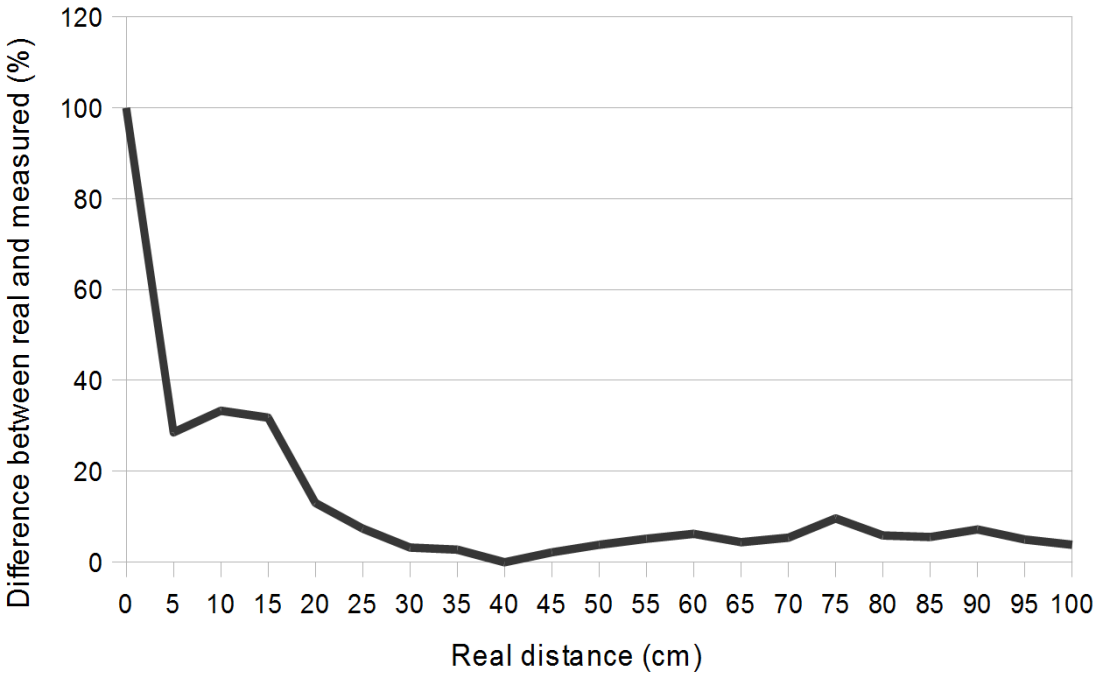

Figure 2.8: Difference between the actual measurements and sensor readings

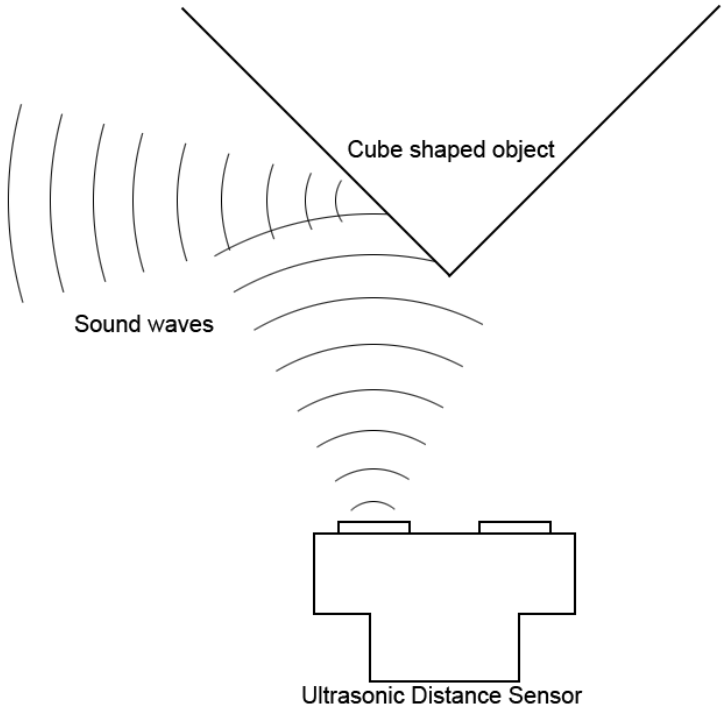

Figure 2.9: Cubes and the reflections of sound waves

able to change direction. Considering the measuring range of the sensor, the target area is very limited, as seen in figure 2.10.

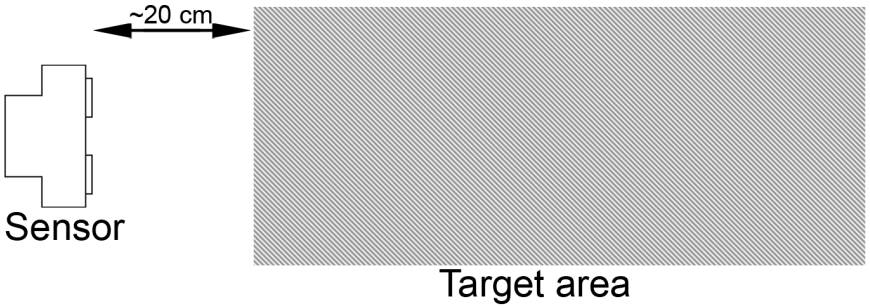

Figure 2.10: Target area of ultrasonic sensor

2.7.2 Infrared Distance Sensor

Goal

The goal of this experiment was much the same as the ultrasonic sensor experiment. This sensor emits an infrared beam. When the beam is reflected back towards the sensor, the hardware will be able to calculate the distance to an object. Since light waves work much the same way as sound waves, different shapes and surfaces impact the sensors ability to detect and measure the distance to an object.

Description

This experiment was performed by moving a box and the IRBall at various distances from the sensor, and measuring the actual distance between the object and the sensor, then comparing it to the distance read by the sensor.

Setup

This setup was very similar to the ultrasonic sensor experiment setup seen in section 2.7.1. The ultrasonic sensor was replaced with the infrared distance sensor, and the IRBall was replaced with objects of different shapes and sizes. Boxes and different LEGO constructions was moved in front of the sensor to see how the readings varied.

Results

The table 2.8 contains the results from the test, and it only shows the test with objects other than the IRBall. It is impossible to get any meaningful measurements using the IRBall. It is due to the signals emitted from the IRBall that interferes with the infrared receivers in the distance sensor. The IRBall emits a pulse of infrared light with a certain frequency and this disrupt the operation of the distance sensor.

The first column is the actual measurements and the second column is the output from the sensor. The raw data was used to produce the graph seen in figure 2.11. It shows the

difference between the actual distance and the measured distance in percentages. The graph shows that when objects are very close to the sensor the measured distance is off by up to 25%. From 10 cm to about 45 centimetres the deviation is under 5%. Further out the deviation rises and varies around 10%. There does not seem to be a trend line of any sort.

Actual (mm)	**Sensor (mm)**
50	70
100	97
150	152
200	200
250	249
300	301
350	367
400	419
450	462
500	564
550	620
600	676
650	707
700	765
750	860
800	860
860	900

Table 2.8: Readings from the optical distance sensor experiment

Sources of errors

- The sensor can be affected by its surroundings, e.g. sunlight.

Conclusion

Between 10 and 40 cm the infrared distance sensor is fairly accurate. When this distance is exceeded, the accuracy decreases. When trying to measure the distance to the IRBall problems were encountered.

This means that the infrared distance sensor could not be used to measure distance to the target if the target carries the IRBall. The sentry could only be built using either the IRBall or the optical distance sensor. This means that the infrared distance sensor, even though it is accurate, could not be used to built the sentry as the only way to track the target was to use the IRBall.

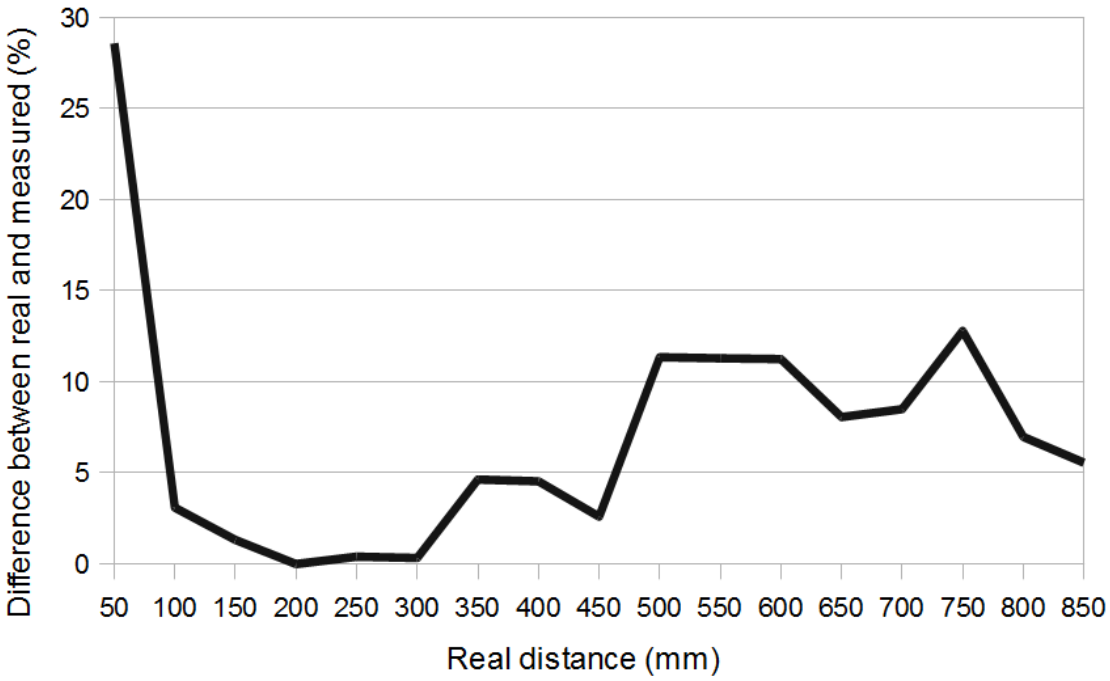

Figure 2.11: Difference between actual measurements and sensor readings

2.7.3 IRSeeker Test

Goal

The goal of this experiment is to determine how the infrared seeker works. Inside the seeker, there are several infrared receivers. Each of these receivers picks up signals at different angles from the seeker. These angles will be measured. As a source of infrared light the IRBall is used.

Description

The IRSeeker was placed on top of a horizontal blackboard, then the IRBall was moved around, and lines were drawn whenever the sensor output changed. A small program was created that made the NXT Brick play a short sound whenever this occurred. The ball was then moved around the sensor in varying distances. When the NXT Brick played the sound, a mark was set at the position of the ball. A sketch of this setup can be seen in figure 2.12.

Setup

Figure 2.12 shows a drawing of the setup for this experiment. Again, as with the previous experiments, the IRSeeker is positioned at a stationary position and the IRBall is then moved around the IRSeeker. The IRBall is only moved in two dimensions, the omitted one being the vertical direction. A picture of the setup can be seen in figure 6.3 in appendix A.

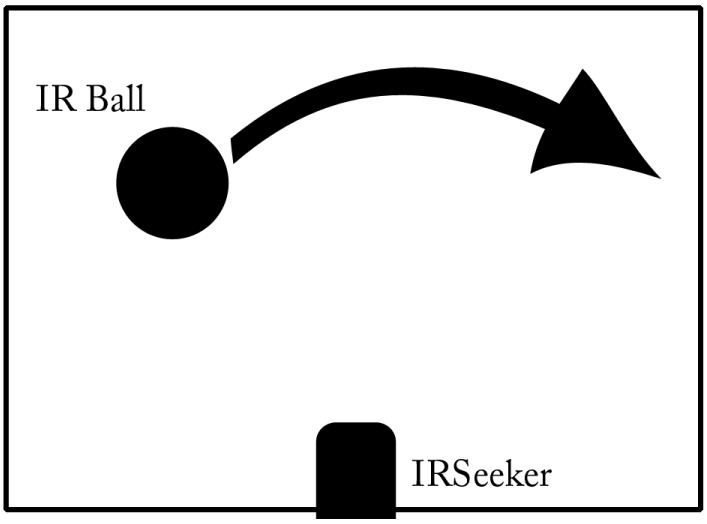

Figure 2.12: A sketch of the setup for the IRSeeker.

Result

The results are shown in table 2.9. In this table the first column is the field number, which the IRSeeker gave as output, and the next column is the angle that has been calculated for that specific field. All of the results are illustrated in figure 2.13 in order to give a good overview of the width and the location of the fields.

Field	**Angle**
1	43,43°
2	8,03°
3	45,91°
4	10,04°
5	56.07°
6	8,6°
7	46.53°
8	8,03°
9	46,53°

Table 2.9: Readings from experiment with the IRSeeker sensor.

Sources of errors

- The IRBall itself is inaccurate because it is not possible to know which infrared LED the IRSeeker is tracking.
- The assumption that the division of fields are linear is not easily verified.
- The IRSeeker is affected by all infrared light, including that of sunlight, artificial light etc.

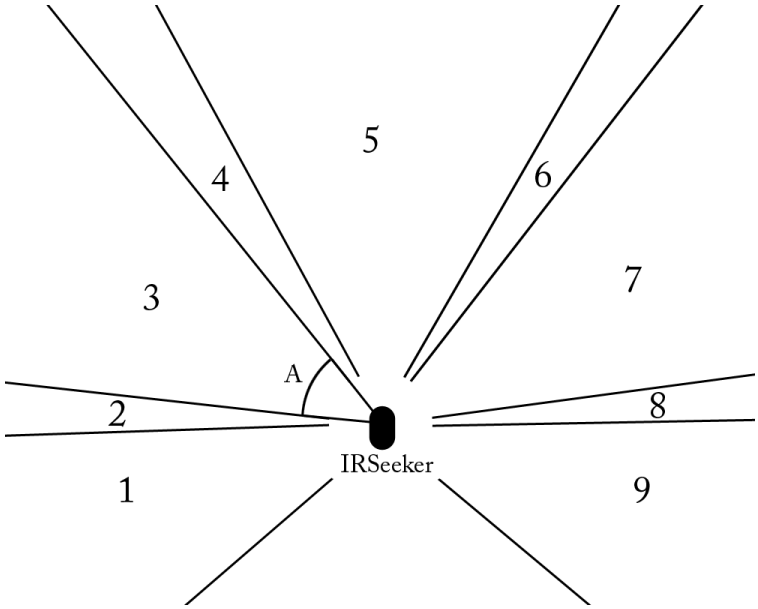

Figure 2.13: The different fields of the IRSeeker. The number in the field is the output given by the sensor.

Conclusion

The experiment clearly shows the different sensors in the IRSeeker. Field 1, 3, 5, 7 and 9 correspond to each of the five infrared sensors inside the IRSeeker. During the experiment it was discovered that fields were overlapping. These are marked 2, 4, 6 and 8, these fields are narrower than the rest. The angles of the different fields were calculated using the law of cosine. The results indicate that the fields do not have the same angles, but there is some symmetry. For example, fields 3 and 7 are both close to 46°in width and fields 2 and 8 are both close to 8°.

An example use of the IRSeeker could be the scenario where one of the narrow fields is used to locate the IRBall, because it is narrower it will be possible to track the IRBall more accurately. The IRSeeker has to be turned in order to keep track of the IRBall. If the IRSeeker was stationary, it would still be able to track the IRBall, but not as accurately, since the IRBall would have the possibility of passing the wider fields 1, 3, 5, 7 and 9.

2.7.4 Summary

After having experimented with the sensors, it was possible to conclude which sensors was best suited for what task. The infrared distance sensor does not function together with the IRBall, the ultrasonic sensor performs badly against spheres and pointy shapes and the IRSeeker only works together with infrared sources. The accuracy of each device was analysed and with this data it will be possible to exclude the sensors not suited for this project and keep the ones that are.

2.8 Problem Statement

The analysis covered the general theory about real-time systems and the embedded hardware these systems can be implemented on. From here, choices about the sentry implementation must be made based on this theory. The following lists of questions about the sentry needs to be answered.

- Which hardware platform is most ideal for the automatic sentry?
- Which sensors and actuators are needed?
- Which scheduling algorithm is best for for the automatic sentry?

2.8.1 Project Limitation

To make design and implementation of IRIS possible within the time limitations of this project and the given hardware limitations there has to be some limitations to the robot as well. The sentry will only be developed so that is able to predict the movement of a target moving at steady speed in a straight line, in close proximity to the sentry. There will only be time for implementation of a single scheduling algorithm, however other algorithms were discussed for use. Finally only a single hardware platform will be used for implementation.

Chapter 3

Design

The analysis of theory concerning RTS, embedded systems, the JOP and the results from the experiments will provide the basis for this chapter. Based on the gathered information design ideas for the robot are produced. Prototypes of the robot will be explained, discussed and taken into consideration in this chapter, so a final design choice can be made for the construction of the robot and the development of the software.

3.1 Overall Design Concepts

In order to solve the initiating problem, the system must fulfil certain requirements. The objective is to shoot a moving target with a constant velocity. Since it is not possible to build a system that rotates a cannon and fires a projectile infinitely fast, a future position of the target is required to be predicted at the time when the cannon is ready. The velocity of the target must be known for this to be possible, so in order to calculate the velocity of the target, multiple positions of the target must be recorded. The velocity and direction of the target can then be calculated using basic physics, vector mathematics and trigonometry. Considering that the target is moving, the system is required to make the calculations that determine in which direction the sentry is to fire, before the target is out of range.

The following sections will describe two proposed designs for the robot.

3.1.1 Single Station

In this design the following components are used:

- 1x IRSeeker Sensor
- 1x Ultrasonic Sensor
- 1x Servo Motor for turret rotation

- 1x Servo Motor for firing mechanism

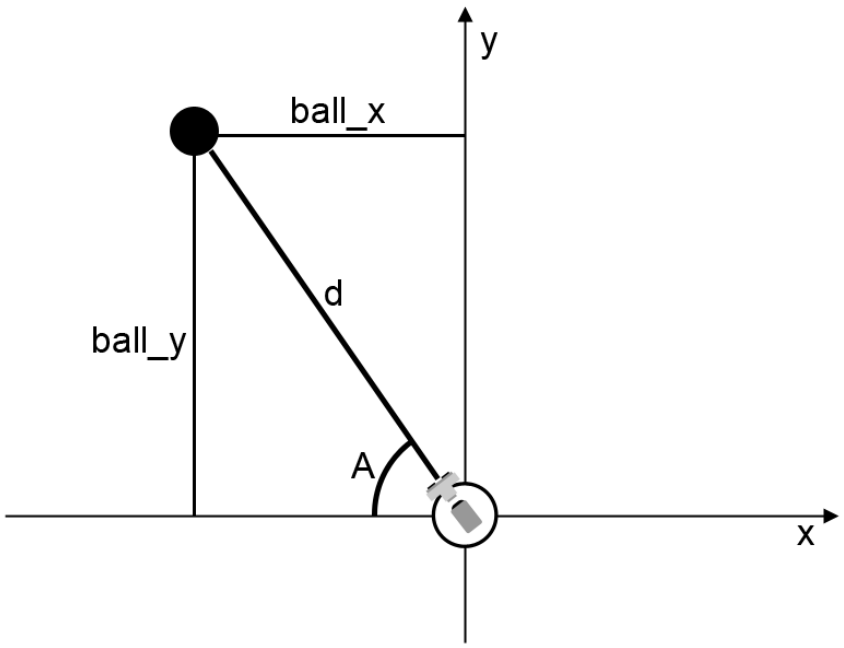

Figure 3.1: Single station design

The IRSeeker is used to find the angle to the IRBall (the target), the ultrasonic sensor is then rotated with the rest of the sentry. The ultrasonic sensor should point towards the IRBall so that the distance may be measured.

Only one of the narrow fields are used from the IRSeeker, for instance field 6, which can be seen in figure 2.6, and the sensor is rotated along with the station by a servo motor. This method makes it possible to look in 360°, instead of just the field that the IRSeeker can seen, which is smaller. Using field 6 combined with rotation, it is possible to view all 360°.

If this distance is within the 100 cm limit in which the sensor can measure, the position of the target is calculated using simple trigonometry. A short period of time afterwards, a second position is calculated using the same method. These two positions are used to determine the velocity vector of the moving object. At last the cannon, which is mounted on the servo motor along with the sensors, is turned to the angle where the object will be at firing-time, and finally a projectile is fired. In figure 3.1 a concept of the sentry is seen from above. The black dot is the IRBall. In the figure, bold lines represent the two readings taken to be able to find the position of the target. The angle A is measured using the tachometer of the motor. The angle returned is the number of degrees the motor has turned since its last reset. The distance d is measured by the ultrasonic sensor. The x-axis represent the direction in which the sentry was turned when the last reset occurred. It does not matter what direction the x-axis is in, because the sentry calculates all positions relative to the same x-axis defined at the motor reset. When A and d are known it is possible to calculate

ball_x and ball_y which is the position of the IRBall in relation to the sentry.

3.1.2 Two Stations

In this design the following components are used:

- 2x IRSeeker Sensor
- 2x Servo Motors for sensor rotation
- 1x Servo Motor for firing mechanism

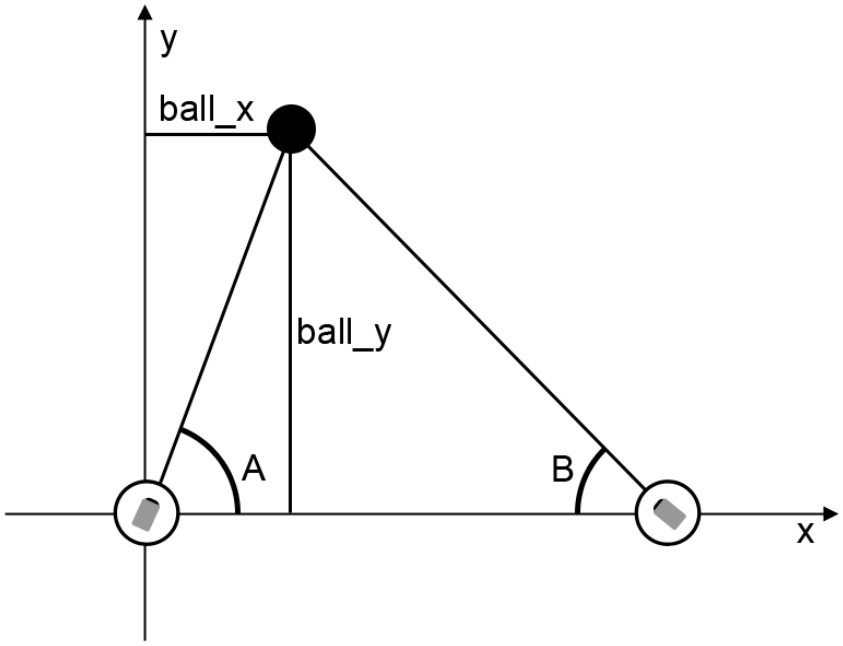

Figure 3.2: Two station design

In this setup, which can be seen in figure 3.2, the two IRSeekers was mounted on a servo motor each, placed at a certain distance from each other. Given these two angles and the distance between the two stations, the final angle and the other two sides could be calculated using the law of sines, thus the position of the object was also calculated. At last the cannon, which was mounted on one of the servo motors along with the IRSeeker, was directed at the angle where the object will be at firing-time, and finally a projectile was fired.

3.1.3 Comparison

In order to find the advantages and disadvantages of the two designs an experiment was conducted.

Goals

The goal of the experiment is to determine which of the two designs is the most accurate for finding positions of the target.The designs in question are the single station and the two station scenarios.

Setup

The setup can be seen in figure 3.2.

Description

The target and the two stations form a triangle, which can be seen in the figure 3.2. The servo motors together with the IRSeekers made it possible to read the two angles A and B. Since the distance between the two stations is a constant, it is possible to apply the law of sines and thereby calculate the remaining parts of the triangle: A, B and IRBall. This information made it possible to calculate the position of the ball (ball_x, ball_y) relative to the stations. The following shows how the two station design calculated the distance from the ball to one of the stations. To calculate this distance the mathematics shown in equation (3.1) was used.

The following formula will calculate the distance to the target from the left station.

$$d = \frac{\sin(B) \cdot dbs}{\sin(180^\circ - A - B)} \tag{3.1}$$

Where d is the distance from the left station to the target, dbs is the static distance between the two stations, A is the angle at the left station and B is the angle at the right station.

Both designs make it possible to calculate the position of the IRBall but which one is the more precise? In order to compare the two designs, the calculated distance from the ball to the sentry from each design, was compared. The single station design uses the distance d it measures with the ultrasonic sensor and the two station design will use a calculated distance to one of the stations.

Because the two robots measure angles in the same way using the motor tachometers, only the distance was compared. As the single-station robot uses the ultrasonic sensor to determine distance, and an experiment with the ultrasonic sensor has been already conducted, see section 2.7.1. This means that it was not necessary to conduct a new experiment with the single station design, as the results from the experiment with the ultrasonic sensor could be compared to the new experiment with the two station design.

To obtain data about the two station design, the IRBall was placed in various positions

in front of the stations. The distance between the station and the ball was measured, for each measurement the calculated distance returned from the program was noted as well.

Result

The results can be seen in table 3.1, this table shows raw measured values in cm. The position of the IRBall was varied, both in distance and direction.

Actual (cm)	Sensor (cm)
24	38
24	23
47	37
97	66
80	107
42	58
40	49
58	56
80	76
70	117
46	93
50	73
62	78
72	98
66	128

Table 3.1: Readings from experiment with the two-station-robot

The results from this experiment are compared with the results from the ultrasonic sensor experiment in figure 3.3. In this figure the deviation between the actual measurements and the reading from the experiments are shown in percentages.

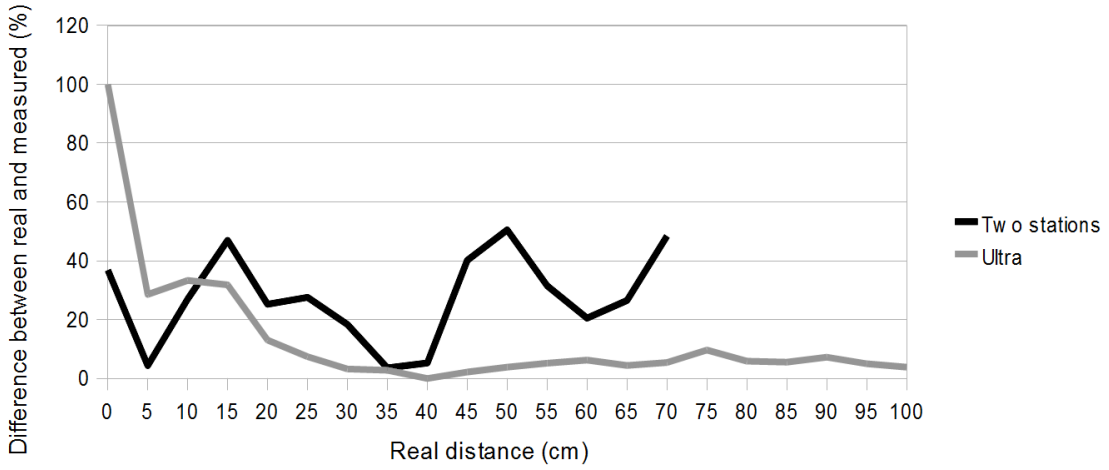

Figure 3.3: Difference between actual measurements and sensor readings

Sources of errors

- All the sources of error from the experiment with the single IRSeeker are also valid in this.
- The servo motor has a tendency to give a faulty angle when turning a large number of times.

Conclusion

This experiment shows that the ultrasonic sensor is more precise at distances above 10 cm, while distances measured by the two station robot vary a great deal. The ultrasonic sensor is however limited to one meter where the IRSeeker can detect the ball meters away. This means that if there is a clear line of sight, the two station design could calculate positions at a greater range.

When the IRball would enter the area between the two sensors, the two station design tended to provide more accurate measurements, as opposed to when it was outside said area.

3.1.4 Choice of Design

The single-station robot is the most simple robot and according to the experiment, it is also the most precise.

Another disadvantage of the two station design is that it is harder to reset the tachometers. The one station design can always reset its tachometer as long as there is no coordinates saved. It "decides" where the x-axis is. The two tachometers of the two station design need to be adjusted to the same x-axis. This has to be done by turning the two motors turn to a certain point where they, as an example, trigger a sensor that reset their tachometer.

3.2 Choice of Platform

Given the choice of sentry design, an acceptable hardware platform must be chosen. The JOP has advantages in terms of WCET-analysis. It is possible to perform very accurate WCET-analysis by decompiling programs and counting bytecode instructions. The reason for this is that WCET for each bytecode instruction is known for the JOP, which is not the case for the NXT Brick. The NXT Brick was not developed with the intention of easy WCET analysis. This means that WCET for tasks has to be measured rather than calculated for the NXT Brick.

When it comes to sensors, the NXT Brick has the advantage, as the JOP library only supports use of the old RCX actuators and sensors, which means that one would have to

write code to support the NXT sensors. One has to alter the LRBJOP to interface with the NXT sensors.

Given these circumstances, the choice is the NXT Brick, because the design depends on the IRSeeker sensor, the servo motors and the ultrasonic distance sensor.

3.2.1 Choice of NXT Firmware

Since the choice of platform is the NXT Brick, the firmware for the NXT must also be chosen. It was decided that the choice of programming language was Java. The reason for this was that the group had experience with Java from earlier semesters and that the focus of the semester would not be to learn a new programming language. This is the main reason why the choice was LeJOS NXJ.

Besides that LeJOS NXJ has large community and the API is well documented. By far the most NXT sensors are supported in LeJOS NXJ, either because they are in the LeJOS library, or because the manufacturers of the sensors have made their own classes extending LeJOS classes.

3.3 Tasks

The structure of the software will depend on the tasks the system must execute. This section will describe the tasks. The Single Station design uses an IRSeeker to find the angle of the target combined with an ultrasonic distance sensor to find the distance to the target.

The following list gives a rudimentary understanding of how the robot is supposed to work.

1. Get data from sensors (IRSeeker, Ultrasonic and Tachometer).

2. Turn based on data. If no predicted position has been found, the sentry will turn after the target. If a predicted position has been found, it will turn towards that position.

3. Calculate the current position of the target.

4. Calculate a predicted position of the target.

5. If a predicted position has been determined and the robot points towards it, the robot will shoot.

3.4 Choice of Scheduler

Before implementing a scheduler for the sentry, a scheduling algorithm has to be chosen, and the scheduler has to fit the tasks of the system.

For various reasons the tasks have to be implemented to some degree before an actual scheduler could be developed. During this implementation it was discovered that all of the tasks could be created such that they were all periodic, i.e. no aperiodic or sporadic tasks. Since sporadic and aperiodic tasks can be avoided, there is potential for implementation of a cyclic executive.

A disadvantage of the cyclic executive is the difficulty of incorporating tasks with a long execution time.

Long execution time means that the execution time is close to the period of a task.

Considering that the system of this project does not have any tasks that block the system or have too long an execution time, cyclic executive is the logical choice for this project.

The sentry with its current tasks may be implemented using either the FPS algorithm or the cyclic executive. With FPS it will be possible to handle sporadic and aperiodic tasks more efficiently. This means that the shoot task could be programmed as a sporadic task and not a periodic one. Therefore it would not be necessary to execute the shoot task until the predicted position of the target is found. In a cyclic executive it would be necessary to make time for the shoot task, even if there is no target. By using FPS the time is used more efficiently, as only the necessary tasks will be executed.

Another thing to take into consideration is memory. Since the cyclic executive runs in a single thread all tasks have access to the same variables. If a FPS scheduler is used, there would be a thread running for each task meaning it would be more difficult to have the tasks share data. The tasks that calculate the position of the target for example require the data fetched from the sensors.

Depending on the requirements a system may be scheduled using either one. This project does not have any tasks that either require priority assignments, that block each other or have a long execution time, ergo a good choice is the cyclic executive scheduling algorithm.

3.5 Scheduler

In this section the design for the cyclic executive is described according to the given tasks. It will also describe how to divide tasks with the given computation times and periods into minor cycles.

3.5.1 Cyclic Executive Design

In this section the design of the cyclic executive according to the given tasks is explained.

An interrupt or a timer is used to control the timing of each minor cycle. Generally they work the same way, where the system will wait until a certain time and then continue with the next cycle. In the following example a timer will be used.

There will be a method for each task in the system that will contain the code of the task. In the following example four tasks are defined in table 3.2. Each task has a predefined period and a computation time. These values are defined or calculated before runtime and cannot be changed.

Task	Period, T	Computation Time, C
A	50	10
B	50	10
C	150	20
D	150	20

Table 3.2: Task used in the cyclic executive design

A system with the four tasks can be scheduled like in listing 3.1.

```
1  while(true) {      // Start major cycle
2    Time = 0
3    taskMethodA();     // Start minor cycle 1
4    taskMethodB();
5    Wait for time = 50
6    taskMethodA();     // Start minor cycle 2
7    taskMethodB();
8    taskMethodC();
9    Wait for time = 100
10   taskMethodA();     // Start minor cycle 3
11   taskMethodB();
12   taskMethodD();
13   Wait for time = 150
14 }
```

Listing 3.1: Pseudo code for a cyclic executive scheduler

The major cycle has a duration of 150 time units while the minor cycles have a duration of 50 time units. In a cyclic executive there is no notion of threads for the tasks, but a list of method calls timed in a loop. Additional protection can be added to check whether a method has used more time than it was supposed to. A boolean flag can be set in the very

end of the last method in a minor cycle and the timer can check this flag before it starts the next minor cycle. This will allow the system to identify that one of the methods in a minor cycle has had a time overrun.

Chapter 4

Implementation

This chapter covers how the robot was built in LEGO and how the software was implemented. All of the important tasks will be described from turning the sentry to predicting the targets position. Limitations and issues of the robot are discussed to give an overview of the capabilities of the robot. The method of measuring the WCET is also described, and these results are used in a utilization test and a RTA, in order to see, whether it is possible to build an FPS for the IRIS.

4.1 Robot Description

This section contains a brief description of the construction of the two robots, IRIS and the target robot.

4.1.1 IRIS

The design of IRIS has been created to be the simplest possible implementation of the concept of the single-station design. It needs to have an ultrasonic sensor and an IRSeeker, some sort of firing mechanism and it has to be able to point all of these in any angle.

For sentry to be able to turn 360°, the sentry consists of a base and a body. The base is the support on the floor, and the body contains the NXT Brick, the sensors and the firing mechanism. A servo motor mounted on the body turns the joint between the base and the body. This will prevent the wire between the servo motor and the NXT Brick from being tangled up in the joint.

The ultrasonic sensor is pointed outwards and is mounted to front of the sentry. The IRSeeker is mounted on top of the NXT Brick for the largest possible field of view. It is angled in such a way that the field with number 6 is pointed in the same direction as the ultrasonic sensor. The reason for this is that the field number 6 is one of the narrow fields,

and therefore the robot will be able to determine the position of the target more accurately with one of the narrow fields than with the wide fields.

Finally a servo motor and two LEGO TECHNIC Competition Cannons are combined in such a way that the cannons can be fired individually. The cannons are pointing forward aligned with the ultrasonic sensor.

4.1.2 Target Robot

The target robot also has to fulfil some requirements for IRIS to work. It has to move in a straight line, at a steady speed and it has to emit infrared radiation which the IRSeeker can detect. Another NXT robot is build for this purpose. This is a simple robot consisting of two motors for driving the wheels and a socket for the IRBall to be placed in.

Pictures of IRIS and the target can be seen in appendix A. (Figure 6.1 and 6.2)

4.2 Tasks

The general strategy for implementation is to make the tasks as small as possible. For instance if input from a sensor is needed, there is a task that only performs this operation.

Since it was decided that the scheduler would be cyclic executive, the simplest way to implement the individual tasks would be by containing their individual code in static methods.

4.2.1 Get Direction

Get Direction, which can be seen in listing 4.1, overwrites the *irSeekerDirection* variable with the direction of the infrared ball with a number ranging from 1-9 according to the fields of the IRSeeker or 0, if the ball is not detected. The *irSeekerDirection* variable is a static integer that is used to store the result of this task.

```
1 public static void getDirection() {
2   irSeekerDirection = seeker1.getDirection();
3 }
```

Listing 4.1: The getDirection task

4.2.2 Get Distance

GetDistance, which can be seen in listing 4.2, overwrites the *ultrasonicDistance* variable with the distance from the ultrasonic sensor to the object ahead as an integer representing the distance in cm. 11 cm are added because the sensor is placed 11 cm from the center of the sentry.

```
1 public static void getDistance() {
2   ultrasonicDistance = sonic.getDistance() + 11; // + 11 cm for distance to
        the center of the turret.
3 }
```

Listing 4.2: The GetDistance task

4.2.3 Get Tachometer-count

getTachometerCount, which can be seen in listing 4.3, overwrites the *motorTachometerValue* variable with the current tachometer-count of the motor that rotates the sentry.

```
1 public static void getTachometerCount() {
2   motorTachometerValue = motor.getTachoCount();
3 }
```

Listing 4.3: The GetTacho task

4.2.4 Turn Sentry

In the TurnSentry task there are three distinct branches through the code. The first branch is to turn the sentry at a steady speed in case that the target is not in sight. The purpose of this is to make up for the blind angle of the IRSeeker, which is shown in figure 2.13. By turning the sentry the target cannot keep hiding in the blind angle, because the sentry will eventually check in every direction.

The second branch is used to turn the sentry towards the target in order to point the ultrasonic sensor at the target. The sensor needs to point at the target as fast as possible, but if the speed of the motor is set too high, the movement becomes very inaccurate and will most likely be turned too far, as the motor will not be able to stop in time. This issue is handled by using the IRSeeker. The program simply switches the turn speed based on the output of the IRSeeker. If the sentry has to make a wide turn in order to point at the target, the speed will be set high, but if the sentry only has to turn a little, the speed is set low. In this way it turns more accurately than if it turned at full speed all the time.

In the final possible branch, which can be seen in listing 4.4, the sentry turns toward the target position which has been calculated. The tachometer will not reset to zero, if it passes 360°, instead it will keep adding the number of degrees to the same variable. In listing 4.4, line 4 the angle is reduced to the corresponding number between 0 and 360°using modulus, in order of preventing the sentry turning an unnecessary number of times. The statements in lines 6-23 are responsible for turning the motor towards the predicted angle of the target with a 10°tolerance. Lines 6-15 determine which way the sentry must turn in order to ensure it has a maximum turning of 180 degrees. Lines 16-20 turns the motor.

```
1  // Turn after target
2  motor.setSpeed(100);
3
4  motorTachometerValueMod = (int) motorTachometerValue % 360;
5
6  if (Math.abs(motorTachometerValueMod - target.getAngle()) > 10) {
7    forward = true;
8    if (target.getAngle() > motorTachometerValueMod) {
9      if (target.getAngle() - motorTachometerValueMod > 180)
10       forward = false;
11   } else {
12     if (motorTachometerValueMod - target.getAngle() <= 180) {
13       forward = false;
14     }
15   }
16   if (forward) {
17     motor.forward();
18   } else {
19     motor.backward();
20   }
21 } else {
22   motor.stop(); // Aiming at target
23 }
```

Listing 4.4: Turning to the predicted target location

4.2.5 Calculate Current Position

The Calculate Current Position task uses the angle and distance retrieved in GetTacho and GetDistance to calculate the coordinate of the target. It will however only calculate the coordinate, if it fulfills two conditions.

- The target must be within range of the distance sensor.

- The distance sensor must be pointing towards the target.

The coordinate is saved in one of two variables based on timing relative to previous recorded coordinates and whether it already has recorded a coordinate. In line 3 the newly recorded position is saved as a vector, which will be used with a timestamp to create a coordinate.

In lines 5-8 coordinate `coord1`, which is one of the positions needed to predict the future position of the target, is overwritten with the new target position, unless the position is already set. If `coord1` is already recorded, lines 9-12 determines whether that coordinate is too old. Is `coord1` too old according to its timestamp, it is overwritten with the newly recorded position.

Finally coordinate `coord2` can be written, as long as `coord1` has been recorded and is not too old. Lines 13-17 write the coordinate `coord2` and confirm on the timestamp that the new coordinate is not too young, this is important as the two coordinates may not be positioned too close, as it can interfere with the accuracy of the prediction.

```
1  public static void calculateCurrentPosition() {
2    if (ultrasonicDistance > 5 && ultrasonicDistance < 100 && irSeekerDirection
       == 6) {
3      newPos.setVector(ultrasonicDistance,motorTachometerValue);
4
5      if (!isCord1Set) {
6        coord1.setCoordinate(newPos, clock.elapsed());
7        isCord1Set = true;
8      }
9      else if (clock.elapsed()- coord1.timestamp >= 3000) {
10       // coord1 is too old
11       coord1.setCoordinate(newPos, clock.elapsed()); // old coord1 is replaced
             with current one
12     }
13     else if (!isCord2Set && (clock.elapsed() - coord1.timestamp >= 2000)) {
14       // coord1 is not old enough
15       coord2.setCoordinate(newPos, clock.elapsed()); // coord2 is current one
16       isCord2Set = true;
17     }
18   }
19 }
```

Listing 4.5: The Calculate Target Coordinate Task

4.2.6 Predict Position

The purpose of this task is to predict where the target will be positioned in relation to the sentry when the projectile launcher is ready to shoot. The predicted position of the target is based on two coordinates found in calculateCurrentPosition.

The task will first check if the two needed coordinates have been obtained. In line 3 vector subtraction is used to calculate the vector that represents the movement of the target from *coord1* to *coord2*. In line 6 the difference in time between the two coordinates is divided with the movement vector in order to obtain the speed vector, as length divided with time gives speed. The vector vectorBallSpeed now contains the direction and speed of the target.

The code in lines 8-11 will predict the position of the target two seconds after `coord2` is obtained. Finally the target is set in line 12, and `isTargetSet` will be set to true: This tells the Turn Sentry task that it needs to turn towards the predicted target position. A simple experiment shows that a 180°turn would take 1.85 sec, which is why the `shootDelay` is 2 seconds.

```
1  public static void predictPosition() {
2    if (isCord1Set && isCord2Set) {
3      vectorBall.minusProduct(coord2, coord1);
4      int timeBetween = coord2.timestamp - coord1.timestamp;
5      // v = vectorBall/timeBetween = (cm)/(ms)
6      vectorBallSpeed.setVector(vectorBall.getX()/timeBetween, vectorBall.getY()
           /timeBetween);
7
8      int shootDelay = 2000;
9      vectorBallTarget = vectorBallSpeed;
10     vectorBallTarget.scalarMult(shootDelay);
11     vectorBallTarget = vectorBallTarget.add(coord2);
12     target.setCoordinate(vectorBallTarget, coord2.timestamp + shootDelay);
13
14     isTargetSet = true;
15   }
16 }
```

Listing 4.6: The Predict Target Position Task

4.2.7 Shoot

The shoot task is responsible for firing a projectile when the target reaches the predicted position. Since the turnSentry function will make sure the sentry is facing towards the

predicted position, it is only required to check if the predicted time is equal to the current one.

The shoot task is only executed once every major cycle, which is 150 ms long. It is therefore necessary to make the time window long enough for the shoot task to be executed within the major cycle.

That is why it is checked if the difference between the target time and the current time is less than 80 ms. This is done in line 2 in listing 4.7. The window will be 160 ms ($2 \cdot 80$) and since the shoot method will be run every 150 ms it is assured not to miss the window. Line 4 is the code that tells the motor to rotate. It is invoked with true as the second argument which causes the program not to stop and wait for the motor to turn all the way. The rest of the shoot method will reset the recorded positions and the predicted target position. This will make the sentry ready to record new positions and shoot again.

```
1  public static void shoot() {
2    if (isTargetSet && (Math.abs(target.timestamp - clock.elapsed()) < 80) ) {
3      motor.stop();
4      Motor.B.rotate(90, true);
5      //Resets motor
6      tacA = 0;
7      motor.resetTachoCount();
8      //Reset target
9      target.reset();
10     isTargetSet = false;
11     //Reset coords
12     isCord1Set = false;
13     isCord2Set = false;
14   } else if (isTargetSet && (clock.elapsed() - target.timestamp > 80) ) {
15     //Reset target
16     //Reset coords
17   }
18 }
```

Listing 4.7: The Shoot Task

4.3 WCET Measurement

Before implementing the cyclic executive scheduler a table must be created which contains information about the tasks, as the period and execution time of each task must be known.

To measure execution times the task methods must first be implemented, because changes to a method can mean changes in the WCET for that particular method.

To measure the time required to execute a method two timestamps are saved. One just before invoking the method and one right after. The difference between these two timestamps is an approximation of the execution time.

The sentry will be started and data will be output to a text file on a computer using a bluetooth connection. The execution time for each task will be saved to a temporary variable and output via bluetooth. Even though bluetooth data transfer will take time, it should not have any effect on the measured execution times.

Listing 4.8 shows how time is measured for the getDistance method.

```
1 ...
2 tstart = clock.elapsed();
3 getDistance();
4 tstop = clock.elapsed();
5 ...
6 ttmp = tstop - tstart;
7 RConsole.print(ttmp + ",");
```

Listing 4.8: Example of measuring execution time of a task

In line 2 and 4 two timestamps are saved using the `clock` object which is an instance of the LeJOS StopWatch class. The difference between these timestamps is calculated and sent to the computer.

This makes it possible to see how the execution times vary, as the sentry is running. The sentry is started and the target robot is placed in front of the sentry several times at different distances and speeds, in order to make the methods run through their longest execution braches.

An example of the data collected for a task can be seen in figure 4.1. This is a graph produced from over 6000 executions of the method. It is seen that the method most of the times vary from 1 to 7 ms, however some measurements take as much as 17 ms. The reason for this can be the ultrasonic sensor, because it is unknown when the sensor will return a usable value to the system. However the only choice is to acknowledge 17 ms as the worst case execution time of the task getDistance.

The other tasks of the system are measured in the same way as the getDistance task. During the development of the cyclic executive, the code in a method was sometimes changed and it was therefore required to measure the execution time of the method again to make sure the scheduler would still work. The final results of the timing tests can be seen in table 4.1. It shows the getDistance tasks as the most time consuming one, and in general, the tasks that receive data from the sensors take considerably more time than the tasks that process the data.

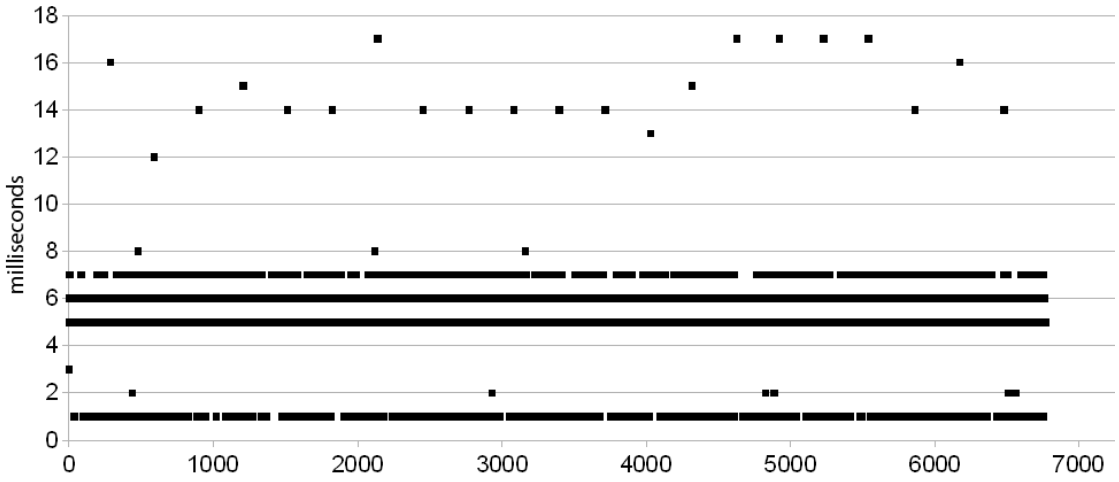

Figure 4.1: Measurements of the getDistance task

Task	Period, T (ms)	Computation Time, C (ms)
getDirection	50	8
turnSentry	50	9
getDistance	50	17
getTachometerCount	50	2
calculateCurrentPosition	50	3
predictPosition	150	2
shoot	150	4

Table 4.1: Tasks of the sentry system

4.3.1 Task periods

The length of each minor cycle is set to 50 ms. The execution time of all tasks sums up to 45 ms and about 10% is as a margin of error. Periods may be configured as desired and therefore some tasks have been chosen to be prioritised more than others. These priorities are not used as in a FPS scheduler, they are used to arrange tasks in order of their relative importance.

The tasks getDirection, getDistance, getTachometerCount, turnSentry and calculateCurrentPosition have a smaller period than the rest, because the tasks predictPosition and shoot will not be required to run as often. If the system has not determined a target to shoot at, these two tasks will do nothing and the scheduler will wait. This is why they have a longer period, and it is seen as a fair exchange to run these tasks less times than the others that execute their code every time. This way there will be less time between each sensor reading and reaction to that value.

PredictPosition and shoot, however, have a fairly low execution time, so measured in time there is not much to be saved. The advantage by running these task fewer times is that for example, getting the infrared reading and turning the sentry will be performed more

times a second, and this helps provide a smoother turning of the sentry. This means that when the sentry is in position to fire and is waiting for the correct time, it will only check the time every 150 millisecond. However it is guaranteed that the sentry will check exactly once every 150 millisecond. If the target moves fast, the 150 milliseconds is a long time, but the system is limited to slower moving targets. If the target moves with a speed of 10 centimeters per second it will move 1,5 centimeter in 150 milliseconds.

4.4 Cyclic Executive Scheduler

This section will show how the cyclic executive scheduler is implemented in IRIS. The data used to build the scheduler is from the previous section. The data is the computation time and period for each task. The implementation of the cyclic scheduler is fairly simple. It consists of three minor cycles, each of 50 ms, which means the major cycle is 150 ms. The tasks with a 50 ms period are executed once every minor cycle while the other tasks with a 150 ms period are executed every third minor cycle.

The major cycle while loop can be seen in listing 4.9

```
1  int startTime = 0;
2  clock.reset();
3  while (!Button.ESCAPE.isPressed()) {
4  // First minor cycle
5  ...
6  while (clock.elapsed() < startTime + 50) {
7    try {
8      Thread.sleep((startTime + 50) - clock.elapsed());
9    } catch(InterruptedException e){
10     continue;
11   }
12 }
13 // Second minor cycle
14 ...
15 while (clock.elapsed() < startTime + 100) { /* Thread.sleep */  }
16 // Third minor cycle
17 ...
18 while (clock.elapsed() < startTime + 150) { /* Thread.sleep */  }
19 startTime += 150; }
```

Listing 4.9: Example of measuring execution time of a task

After the last task of a minor cycle is executed the system must wait until the next minor cycle begins. This could be done by a busy-wait loop that will constantly check time if the next minor cycle should be started. This solution is not perfect because it is a waste of clock

cycles. To save some CPU time the sleep method is invoked on the main thread. This will suspend the thread until the next minor cycle is supposed to start. If the sleep method casts an exception the busy while loop will continue and try to suspend the main thread again. When it is time to start the next minor cycle the while loop should break and allow the cycle to start.

To control the timing of the main while loop in line 3 there is a global integer: *startTime*. For each run through of the while loop this counter is incremented by 150. This counter is used to determine how long to suspend the thread before each minor cycle. In line 6 it is compared to the current timestamp. If the current timestamp is smaller than the time where the next minor cycle starts, the while loop will suspend the thread. In line 8 the remaining time until the next minor cycle is passed as an argument to the sleep method. This will suspend the thread for the correct amount of time. Note that the sleep code is omitted in line 15 and 18. The only difference here is that the second cycle will start 100 ms after *startTime* is incremented and the third after 150 ms.

Having a global time counter will make sure that each minor cycle starts at the correct timestamp. There is however one fatal flaw of this approach. The maximum value of an int in leJOS is 2147483648. When this number is reached the operation of the sentry cannot be guaranteed any longer. Converting the maximum milliseconds to days is estimated to 24 days of operational time. This is acceptable because the batteries will run out before. If the system did not run on batteries, it would have been necessary to reset the sentry after 24 days. Another solution could be to reset the `startTime` and the `clock` object after a certain time.

If a task for some unknown reason suddenly has an execution time longer than the test showed, the system could fail in several ways. The most critical part is when the sentry has turned towards the predicted target position and is waiting to shoot. The shoot task is required to execute once every 150 ms to check if the time for firing is right. If some tasks make it impossible for the shoot task to check every 150 ms the shooting window could be missed. This means that the cannon is not fired. Incorporated in the shoot task is a safety to handle this. If the shoot tasks misses the shooting window the calculated positions and the predicted target position will be reset. This will not correct the error made, but merely make the sentry ready to try again.

If the timing of the cycles is exceeded the scheduler will be able to correct this to a certain degree. If the system is behind schedule there will be no waiting after each minor cycle. This will make the system capable of catching up if the time differences are not too large. The periods of the tasks will not follow the ones defined for the system though. A better solution could be to reset the sentry and start over if the timing becomes incorrect.

4.5 Classes

In this section some of the important classes of the IRIS software will be described and discussed. This should give a better understanding of the system, and how the sensors and motors are controlled. The vector and coordinate classes are included, because these classes contains one of the main points in this system, how the position of the target is perceived.

4.5.1 TurnMotor Class

The TurnMotor class is a wrapper of the Motor class that is built into LeJOS NXJ. The main reason for creating a new class for the motor was to be able to use gearing in the robot, but the class helped solve a few other minor problems as well.

Gearing

When instantiating a new object the gear-ratio is required in the constructor.

```
1 public TurnMotor(Motor motor) {
2   this.motor = motor;
3 }
4
5 public TurnMotor(Motor motor, double gearRatio) {
6   this.motor = motor;
7   this.gearRatio = gearRatio;
8 }
```

Listing 4.10: Constructors for TurnMotor

The code for the constructors is given in listing 4.10, where the first constructor only takes a Motor object as parameter and assumes a gear-ratio of 8/24, which is the number of teeth on the gear system the robot uses. When the gear-ratio is set it is automatically used in all the standard functions from the Motor class that rely on an angle to work correctly, such as *rotate* and *getTachoCount*, the code for *rotate* can be seen in listing 4.11.

```
1 public void rotate(int degrees) {
2   motor.rotate((int)(degrees/gearRatio));
3 }
```

Listing 4.11: The *rotate* function in TurnMotor

4.5.2 Vector

The system requires a way of expressing the coordinate of the target and it is often practical to be able to convert between expressing them with x and y, angle and distance. This is

why a vector class is needed for expressing two-dimensional vectors.

The Vector class is simple, as it only has two variables, an `x` and a `y` double, which can be set with two different types of constructors, as seen in listing 4.12. One constructor takes the `x` and a `y` and the other takes an angle and a magnitude and calculates `x` and `y` from these.

```
1  public Vector(double x, double y) {
2    this.x = x;
3    this.y = y;
4  }
5
6  public Vector(int magnitude, double angle) {
7    this(
8        Math.cos(Math.toRadians(angle)) * magnitude,
9        Math.sin(Math.toRadians(angle)) * magnitude
10       );
11 }
```

Listing 4.12: Constructors of Vectors

The class contains useful methods for doing common vector operations, such as addition, subtraction, dot product etc. It also allows conversion from x and y representation to *angle* and *magnitude* representation.

4.5.3 Coordinate

Coordinate is a very simple class that extends Vector. Its only difference is that it has a `timestamp` member in the form of an integer along with different constructors and a toString method. This is used to associate time with the coordinates the system obtains for the target in order to predict its position at launch time.

4.6 Limitations

The system will be limited to a degree of functionally since the sentry only works under certain conditions.

The sentry is only designed to work in two dimensions. It can only turn around one axis and it is therefore assumed that the sentry and the target is placed on a level surface, e.g. a floor. The ultrasonic sensor points outward from the sentry, this also requires a level surface. The range of the ultrasonic sensor limits the range of the sentry to the gray area in figure 4.2. It is clear that the minimum and maximum range of the ultrasonic sensor defines the area where the sentry is able to track the target. It is also assumed that the area around the

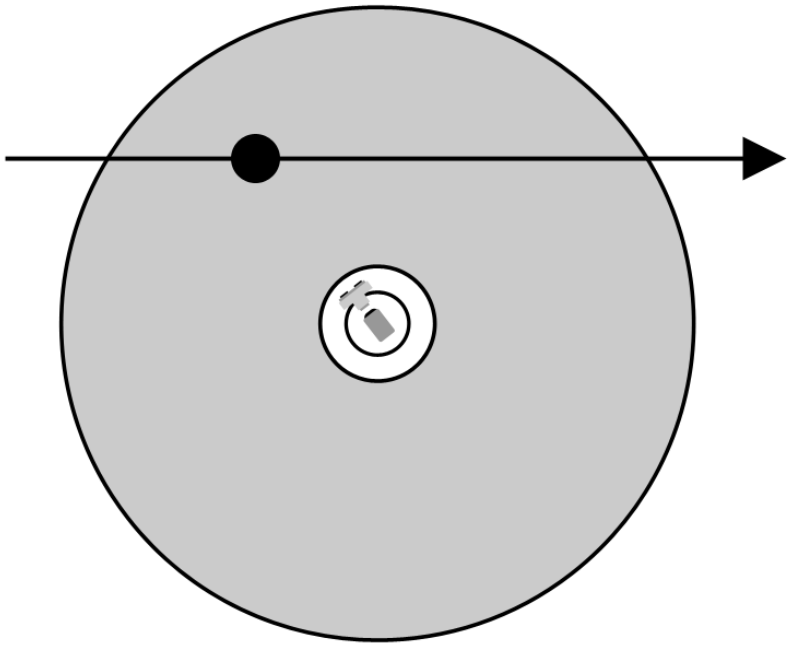

Figure 4.2: The sentry and the target sketched as seen from above

sentry is clear of obstacles because the sentry requires a clear line of sight in order to track the target.

Concerning the target the following assumptions are made:

- First of all the target must emit infrared light in order for the IRSeeker to track it.
- The movement path of the target must follow a straight line along the floor, it must also move at steady speed.
- The trigonometry used for the calculation of the target position is based on triangles with straight edges, meaning that no curves are allowed in the path of object movement.

Depicted in figure 4.2, the black dot is the target that moves along the arrow through the target area of the sentry. There is also a limit on the velocity of the target; the angular velocity of the target in relation to the sentry must be less than or equal to the maximum turn speed of the sentry. This means that the maximum velocity of the target may be greater the further away the target is from the sentry. This is because if the target moves too fast the sentry cannot keep up with it, and therefore it will not receive the readings needed to fire the cannon.

The two coordinates taken of the targets position must be recorded at a predetermined interval, this is because the sensors are too inaccurate to capture the position and velocity of the target, if the interval is too short. The target itself is also fairly large and it will require to travel at least its own length (or at least the width of the target) for the coordinates to have a proper distance between them. The interval between the coordinate readings is set

to two seconds, that means the target is required to stay inside the target area of the sentry for a minimum of two seconds. The target coordinate can be outside the target area since it is assumed that the projectile fired at the target travels further than the range of the ultrasonic sensor.

To summarise the assumptions for the sentry are listed below.

General:

- The system only works in two dimensions.
- The sentry must be located on a level surface (floor).
- The target must be inside the range of the ultrasonic sensor.
- There can be no obstacles in the way of the sentry (Clear line of sight)

The target:

- The target must emit infrared light (IRBall).
- The target must have a steady velocity.
- The target must travel along in a straight line.
- Angular velocity of the target must be slower than the turn speed of the sentry.
- The target must be within range for minimum two seconds.

4.7 Evaluation

As described in section 4.6, there is a number of assumptions made about the target, which must to be tested. To be exact there are two parameters that need further examination. They are:

- The velocity of the target
- The distance from the sentry to the target

To be able to get an approximation of those parameters, there are a number of scenarios that can give some idea of these parameters. The scenarios are;

- The target moves directly towards the sentry.
- The target moves straight past the sentry at different speeds.
- The target moves straight past the sentry at different distances.

Once these scenarios have been tested it is possible to give some approximation of the maximum and minimum velocities and distances where the sentry is able to consistently determine the targets trajectory.

4.7.1 Collision

The first scenario is where the target, if not shot down, is going to crash into the sentry. It is important to know the maximum velocity at which the sentry is guaranteed to determine the trajectory of the target and shoot. With a minimum delay of two seconds between coordinate measurements, a two seconds shoot delay and a maximum distance to the ball of one meter it is possible to calculate the theoretical maximum velocity of the target:

$$\frac{100\ cm}{4\ seconds} = 25\frac{cm}{second} \tag{4.1}$$

In this scenario the target will begin with a speed of approximately $25\frac{cm}{second}$, it will then increase the speed if the sentry is able to shoot it, or lower the speed if the sentry is not. Based on the experiment in section 2.7.1, the maximum distance was chosen to be one meter. The setup of the scenario can be seen in figure 4.3.

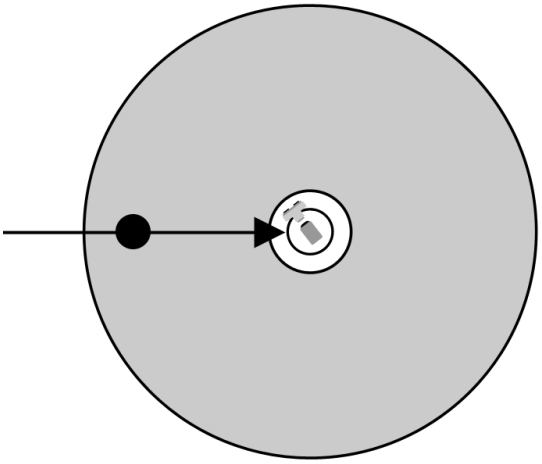

Figure 4.3: The setup of the collision scenario.

Data Analysis

The data collected in this scenario can be seen in table 4.2.

The data shows that the sentry is able to hit a target that moves up to $18.5cm/sec$ directly towards it. Because of the inaccuracy from the LEGO parts used in this experiment, it was not possible to more accurately determine the measurements.

Velocity (cm/sec)	Hit
25.6	No
22.2	No
18.5	Yes

Table 4.2: The speed of the target and whether or not the sentry was able to hit it before collision.

4.7.2 Normal Trajectory

Two scenarios are performed with the target robot passing by the sentry, one at different distances and one at different speeds.

Second scenario

In this scenario a straight trajectory going past the sentry is set. The speed of the target is changed in order to determine the maximum speed the sentry is able to track at a given distance. The speed of the target is increased until the turret is no longer able to shoot the target. Figure 4.4 shows an illustration of the setup, the trajectory denoted by a black arrow going past the sentry, the target is denoted as a black dot.

Third scenario

In this scenario the speed is constanct and the distance to the target is changed. This is performed in order to determine the maximum distance where the sentry can track and hit a target.

The expected result is that the closer the target is to the sentry, the maximum velocity it is able to track becomes smaller.

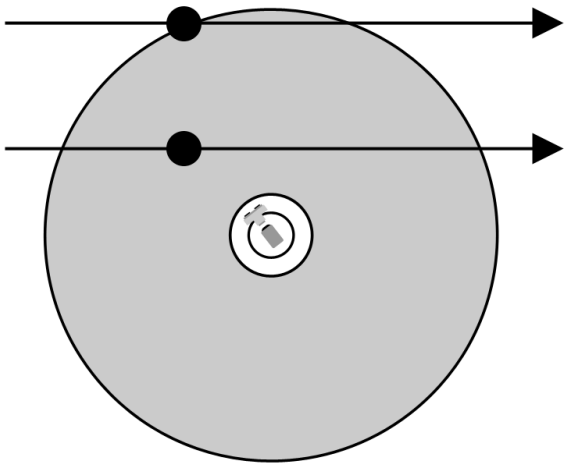

Figure 4.4: The setup of the second and third scenario.

Data Analysis

The two scenarios produce the table 4.3, which contains the number of hits recorded at different velocities and distances.

Velocity (cm/sec)	Hit at 35 cm	Hit at 70 cm
15.6	5/5	2/5
18.2	5/5	1/5
21.7	5/5	1/5
25	5/5	0/5
28.6	5/5	0/5
31.2	2/5	0/5
38.5	3/5	0/5
41.7	1/5	0/5
45.5	0/5	0/5

Table 4.3: The speed and if the shot hit or not are denoted Velocity and Hit accordingly

Table 4.3 shows the maximum speed of the target at a distance of 35 cm, at which the sentry is still able to hit its target. The result of this experiment is that at $28.6 cm/sec$ the hit-ratio is 100 %. If it was enough to hit the target only once, the velocity could go as high as $41.7 cm/sec$. The sentry was however not capable of hitting the target consistently at any speed, if the distance was increased to 70 cm. This is general for all velocities that the sentry becomes very inaccurate as the distance is doubled.

What is not shown in the table is that the sentry is able to determine the location of the target. It was however not capable of hitting the target, because the target was ahead of the projectile. A possible fix for this is an extension of the software that will take the flight-time of the projectile, used for reaching targets at great distances, into account.

4.7.3 Conclusion

To conclude, it is clearly seen that the sentry has the highest hit-ratio if the distance is 35 cm and the velocity of the target is no greater than $28.6 cm/sec$. That said, it is not guaranteed that the sentry can deliver these results again and again. This can be caused both by the inaccuracy of the ultrasonic sensor and the tachometer in the servo motor. Nevertheless the sentry is still satisfactorily accurate in certain scenarios and capable of hitting the target.

4.8 FPS for IRIS

In this section the question whether it is possible to create a FPS for this system with the given data, will be answered based on an utilization-based test and a RTA. The utilization-

based test and the RTA is performed using task parameters described in section 4.3. The data is described in table 4.4, where the first column is the name of the task, the second column is the computation time in milliseconds, the third column is the period for that task, the final column is the priority for that task, when using rate monotonic priority assignment.

Task name	**Task**	**Computation time, C**	**Period, T**	**Priority, P**
Get Direction	a	8	50	7
Turn Sentry	b	9	50	6
Get Distance	c	17	50	5
Get Tachometer-count	d	2	50	4
Calculate Current Position	e	3	50	3
Predict Position	f	2	150	2
Shoot	g	4	150	1

Table 4.4: The set of tasks

4.8.1 Utilization-based test

Utilization-based test is calculated as:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \tag{4.2}$$

The utilization, U, for each task will be calculated and inserted in table 4.5, they are added together in order to compare it with the right side of the equation: $N(2^{\frac{1}{N}} - 1)$. Utilization of the task is calculated as $\frac{C_i}{T_i}$.

Task	**Computation time, C**	**Period, P**	**Priority**	**Utilization, U**
a	8	50	7	0,16
b	9	50	6	0,18
c	17	50	5	0,34
d	2	50	4	0,04
e	3	50	3	0,06
f	2	150	2	0,013
g	4	150	1	0,027

Table 4.5: The set of tasks with utilization

The final calculation is to add these together and compare with $N(2^{\frac{1}{N}} - 1)$:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \tag{4.3}$$

$$0,16 + 0,18 + 0,34 + 0,04 + 0,06 + 0,013+, 0,027 \leq 7(2^{\frac{1}{7}} - 1) \tag{4.4}$$

$$0,82 \leq 0,73 \tag{4.5}$$

The inequality (4.5) is false. This does not mean that the tasks cannot be scheduled, as the utilization-based test can give a negative response on schedulable schemes. The Response Time Analysis is a more precise method.

4.8.2 Response Time Analysis

The RTA is a both necessary and sufficient. If the RTA accepts, the system can be scheduled, if it fails the system cannot. RTA compares the worst response time with the deadline of the given task, so the deadlines for all the tasks are equal to their period. Following is the formula for RTA:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{4.6}$$

The task with the highest priority will have a response time equal to its computation time, and in this system it is task a, meaning that:

Task a:

$$w_a^0 = 8 \tag{4.7}$$

After this the task with the second highest priority must be calculated, which is task b, and etc.

Task b:

$$w_b^0 = 9 \tag{4.8}$$

$$w_b^1 = 9 + \left\lceil \frac{5}{50} \right\rceil 8 = 17 \tag{4.9}$$

$$w_b^2 = 9 + \left\lceil \frac{17}{50} \right\rceil 8 = 17 \tag{4.10}$$

w_b^2 is equal to w_b^1, so 17 is worst response time for task b.

Task c:

$$w_c^0 = 17 \tag{4.11}$$

$$w_c^1 = 17 + \left\lceil \frac{17}{50} \right\rceil 8 + \left\lceil \frac{17}{50} \right\rceil 9 = 34 \tag{4.12}$$

$$w_c^1 = 17 + \left\lceil \frac{34}{50} \right\rceil 8 + \left\lceil \frac{34}{50} \right\rceil 9 = 34 \tag{4.13}$$

$w_c^1 = w_c^2$, so task c will have worst response time of 34, and 34 is less then the deadline of 50.

Task d:

$$w_d^0 = 2 \tag{4.14}$$

$$w_d^1 = 2 + \left\lceil \frac{2}{50} \right\rceil 8 + \left\lceil \frac{2}{50} \right\rceil 9 + \left\lceil \frac{2}{50} \right\rceil 17 = 36 \tag{4.15}$$

$$w_d^2 = 2 + \left\lceil \frac{36}{50} \right\rceil 8 + \left\lceil \frac{36}{50} \right\rceil 9 + \left\lceil \frac{36}{50} \right\rceil 17 = 36 \tag{4.16}$$

$w_d^1 = w_d^2$, so task d will have worst response time of 36, and 36 is less then the deadline of 50.

Task e:

$$w_e^0 = 3 \tag{4.17}$$

$$w_e^1 = 3 + \left\lceil \frac{3}{50} \right\rceil 8 + \left\lceil \frac{3}{50} \right\rceil 9 + \left\lceil \frac{3}{50} \right\rceil 17 + \left\lceil \frac{3}{50} \right\rceil 2 = 39 \tag{4.18}$$

$$w_e^2 = 3 + \left\lceil \frac{39}{50} \right\rceil 8 + \left\lceil \frac{39}{50} \right\rceil 9 + \left\lceil \frac{39}{50} \right\rceil 17 + \left\lceil \frac{39}{50} \right\rceil 2 = 39 \tag{4.19}$$

$w_e^1 = w_e^2$, so task e will have worst response time on 39, and 39 is less then the deadline of 50.

Task f:

$$w_f^0 = 2 \tag{4.20}$$

$$w_f^1 = 2 + \left\lceil \frac{2}{50} \right\rceil 8 + \left\lceil \frac{2}{50} \right\rceil 9 + \left\lceil \frac{2}{50} \right\rceil 17 + \left\lceil \frac{2}{50} \right\rceil 2 + \left\lceil \frac{2}{50} \right\rceil 3 = 41 \tag{4.21}$$

$$w_f^2 = 2 + \left\lceil \frac{41}{50} \right\rceil 8 + \left\lceil \frac{41}{50} \right\rceil 9 + \left\lceil \frac{41}{50} \right\rceil 17 + \left\lceil \frac{41}{50} \right\rceil 2 + \left\lceil \frac{41}{50} \right\rceil 3 = 41 \tag{4.22}$$

$w_f^1 = w_e^2$, so task f will have worst response time on 41, and 41 is less then the deadline of 150.

Task g:

$$w_g^0 = 4 \tag{4.23}$$

$$w_g^1 = 4 + \left\lceil \frac{4}{50} \right\rceil 8 + \left\lceil \frac{4}{50} \right\rceil 9 + \left\lceil \frac{4}{50} \right\rceil 17 + \left\lceil \frac{4}{50} \right\rceil 2 + \left\lceil \frac{4}{50} \right\rceil 3 + \left\lceil \frac{4}{50} \right\rceil 2 = 45 \tag{4.24}$$

$$w_g^1 = 4 + \left\lceil \frac{45}{50} \right\rceil 8 + \left\lceil \frac{45}{50} \right\rceil 9 + \left\lceil \frac{45}{50} \right\rceil 17 + \left\lceil \frac{45}{50} \right\rceil 2 + \left\lceil \frac{45}{50} \right\rceil 3 + \left\lceil \frac{45}{150} \right\rceil 2 = 45 \tag{4.25}$$

$w_g^1 = w_g^2$, so task g will have worst response time on 45, and 45 is less then the deadline of 150.

Table 4.6 shows the results from these calculations with RTA in milliseconds, and it is possible to see that none of the tasks will exceed their deadlines. It is therefore possible to schedule IRIS using FPS.

Task	**Computation time, C**	**Period, P**	**Deadline**	**Priority**	**R**
a	8	50	50	7	8
b	9	50	50	6	17
c	17	50	50	5	34
d	2	50	50	4	36
e	3	50	50	3	39
f	2	150	150	2	41
g	4	150	150	1	45

Table 4.6: The set of tasks with RTA

Chapter 5

Conclusion

The purpose of this project was to examine how real-time systems are developed, and to determine which precautions one must take to assure that the purpose of the system is fulfilled within a given deadlines. To get a practical approach an embedded real-time system was developed. IRIS is an autonomous sentry with the ability to track a moving infrared emitting target and predict the targets future position.

Different scheduling algorithms were examined in depth before implementing the software for the sentry. The cyclic executive was examined as well as the priority based scheduler FPS. The cyclic executive approach was selected for scheduling IRIS because of its simplicity and the small set of purely periodic tasks.

The two hardware platforms that were available, the JOP and the NXT Brick, were analysed. The JOP was more suitable for WCET analysis, but the NXT Brick supported important sensors needed to build IRIS.

In order to thoroughly examine the hardware, a series of experiments were conducted. The accuracy and limits of different LEGO MINDSTORMS sensors as well as some third party sensors designed for the NXT Brick were tested in practice. The result of those experiments was general knowledge about the capabilities and limitations of sensors and actuators.

Two different designs were made for the sentry. These two designs were put into practice to test which one was most accurate for calculating positions of the target. The design with one station was favoured over the design with two stations because of better accuracy. Mathematically the two designs should be equally good at calculating the position of the target, however the deviation between the sensors made a significant difference.

The purpose of IRIS was analysed so it could converted into schedulable tasks. The tasks in the final system were derived from the prototypes and designed to fit in a cyclic

executive. Having implemented the tasks before implementing the scheduler made it possible to measure the WCET of the tasks. This made it possible to construct the major and minor cycles needed in the scheduler.

The scheduler was designed to be implemented in LeJOS NXJ and timing were taken care of using an integrated stopwatch of the LeJOS API. It was possible to modify all tasks to be periodic and fit them into three minor cycles and one major cycle.

Even though only the cyclic executive was chosen for implementation, FPS was also examined, though only at a theoretical level. A utilization-based schedulability test was performed on the set of tasks, which turned out negative. The negative response did not allow any conclusion to whether or not a system is schedulable with FPS. Therefore a full response-time analysis was conducted that turned out positive. This verified that IRIS could be scheduled using FPS.

Building a LEGO sentry able to track a moving target has been achieved. It is able to calculate the velocity of the target and predict its movement ahead in time, though only if the target moves at a slow steady speed in a straight line. This made it possible for IRIS to turn towards the predicted position and wait for the target. When the target moved to the predicted position IRIS fired the cannon.

Chapter 6

Reflection

6.1 Future Development

This section contains ideas for further development of IRIS.

Support for the third dimension could be added in order to shoot targets not confined to a level surface. The NXT Brick is able to handle an additional servo motor and two additional sensors. Adding a third motor would provide the sentry with the ability to tilt along the vertical axis. Another IRSeeker would also be required, so the sentry could track the target upwards. Updated software with the ability to calculate three-dimensional vectors would be required as well. Such an upgrade would be fairly simple to implement, since the mathematics used would be virtually the same for vectors in three dimensions.

The current version of IRIS is limited since it can only cope with targets that move in a straight line. If the target changes directions, the sentry is likely to miss. It would be beneficial if IRIS obtained several target positions and analysed those in an advanced way to recognise different types of patterns of movement.

6.2 Utilization

This section will discuss different applications of a system like IRIS in the real world.

Sentries are often used for military purposes, lethal or non-lethal. It could for instance be placed to guard a certain location.

Using infrared sensitive sensors a sentry could identify intruders in conditions humans cannot, for example at night or in foggy weather. If the location is attacked, no human defenders are risking their lives.

A system being able to detect infrared radiation may have other applications. If the weapons were replaced, the system could be used for more peaceful purposes. Equipped

with cameras it could be a system meant for security and surveillance.

The movement prediction software and infrared sensory could be implemented in cars to assist drivers maneuvering in the dark and predict movement of other cars and avoid accidents. The system would be able to identify sources of infrared light, such as people, large animals or hot engines in other cars.

Another application is fire fighting. Having an automated system to detect and put out fires could save lives without endangering the lives of fire fighters. The system could be sent into burning buildings or used to detect and neutralise wildfires before they spread. Infrared sensors would make it possible to detect fires before they start. For example factory equipment could be monitored for overheating.

In conclusion there is a wide variety of applications for an automated infrared detecting sentry, such as IRIS.

Appendix A: Pictures of IRIS

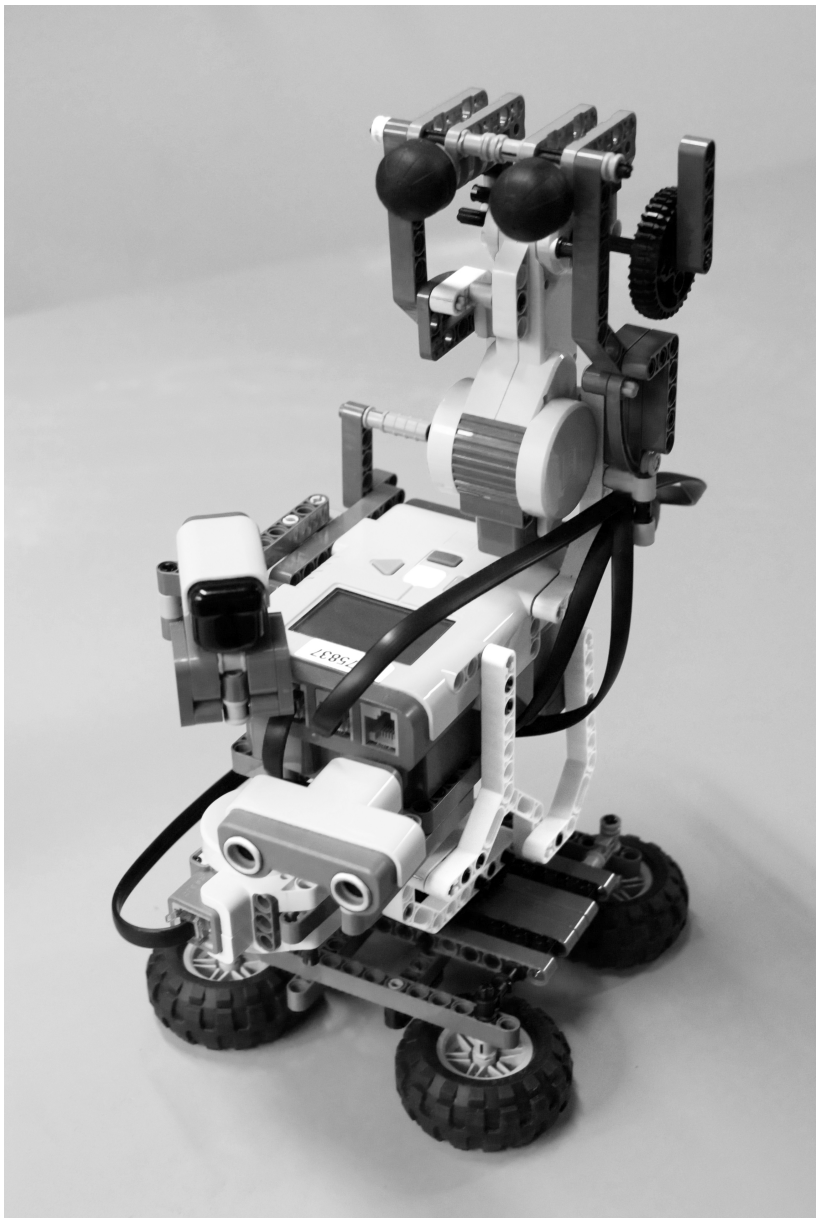

Figure 6.1: Picture of IRIS

Figure 6.2: Picture of the target robot

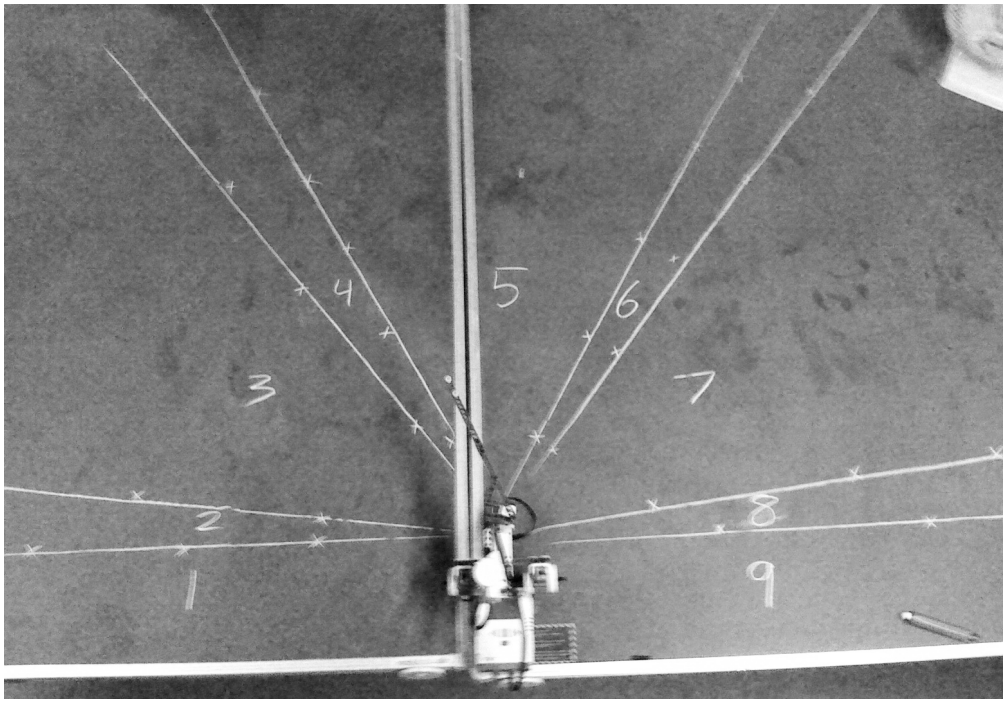

Figure 6.3: Picture of IRSeeker test setup

Bibliography

[1] Alexander Dejaco and Peter Hilber. LRBJOP: A LEGO Robot Controller PCB for the Java Optimized Processor. Master's thesis, Vienna University of Technology, Institute of Computer Engineering, September 2007.

[2] Steve Hassenplug. NXT Programming Software. `http://www.teamhassenplug.org/NXT/NXTSoftware.html`, 2008. [Online; accessed 02-12-2009].

[3] HiTechnic. HiTechnic Infrared Electronic Ball (IRB1005). `https://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=IRB1005`, 2009. [Online; accessed 24-11-2009].

[4] HiTechnic. NXT IRSeeker V2 (NSK1042). `https://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=NSK1042`, 2009. [Online; accessed 24-11-2009].

[5] LEGO. LEGO MINDSTORMS NXT 2.0. `http://shop.lego.com/ByCategory/Product.aspx?p=8547&cn=389&d=292`, 2008. [Online; accessed 11-11-2009].

[6] LEGO. LEGO MINDSTORMS NXT 2.0 servo motor. `http://shop.lego.com/product/?p=9842&LangId=2057&ShipTo=DK`, 2008. [Online; accessed 24-11-2009].

[7] LeJOS. LeJOS. `http://lejos.sourceforge.net/nxj.php`, 2009. [Online; accessed 03-11-2009].

[8] Mindsensors.com. High Precision Medium Range Infrared distance sensor for NXT (DIST-Nx-Medium-v2). `http://mindsensors.com/index.php?module=pagemaster&PAGE_user_op=view_page&PAGE_id=72`, 2009. [Online; accessed 03-11-2009].

[9] Brian Nielsen. NXT HW Sensors and Actuators. `https://intranet.cs.aau.dk/fileadmin/user_upload/Education/Courses/2009/TSW/BNPICS/sw5-es-io.pdf`, 2009. [Online; accessed 21-10-2009].

[10] Dave Prochnow. *The LEGO MINDSTORMS NXT hackers guide*. McGraw-Hill Education, 01-11-2009. page 113, 116, 117.

[11] Anders P. Ravn. Real-Time Systems - The Big Picture. `https://intranet.cs.aau.dk/uploads/media/TSW11.pptx`, 2009. [Online; accessed 11-11-2009].

[12] Martin Schoeberl. WCET Analysis. `http://www.jopwiki.com/WCET_Analysis`. [Online; accessed 10-12-2009].

[13] Martin Schoeberl. A Java processor Architecture for Embedded Real-Time Systems. *Journal of Systems Architecture*, 2008.

[14] Martin Schoeberl. JOP - Java Optimized Processor. `http://www.jopdesign.com/`, 2008. [Online; accessed 28-10-2009].

[15] Martin Schoeberl. A Time Predictable Java Processor. *Design, Automation, and Test in Europe*, 2009.

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