MACROBOT

A Framework for Adaptive Robots
with LEGO MINDSTORMS

Kaspar von Gunten

Diploma Thesis
August 23rd - December 22nd, 2002

Supervising Professor: Prof. Gustavo Alonso
Supervising Assistant: Andrei Popovici
For Dominique
# Contents

Preface .......................... 1

1 Introduction .................. 3
   1.1 Motivation .................. 3
      1.1.1 Assignment .............. 3
      1.1.2 Example ................ 4
   1.2 Robots .................... 6
   1.3 Adaptation ................ 7
   1.4 LEGO MINDSTORMS .......... 7
      1.4.1 Parts .................. 7
      1.4.2 Prototyping ............ 8
   1.5 Programming Environment .. 8
      1.5.1 Rcx Code ............... 9
      1.5.2 Alternative RCX Programming .... 9
      1.5.3 LeJOS ................. 10
   1.6 Aspect Oriented Programming ... 11
      1.6.1 Prose .................. 12

2 Design ......................... 13
   2.1 Requirements Analysis ...... 13
   2.2 Macros .................... 15
      2.2.1 Macro Programs .......... 16
   2.3 Sensitivity ................. 17
      2.3.1 Handlers and Parameters ... 17
   2.4 Brain and Spine (horizontal cut) ... 19
      2.4.1 Brain ................. 20
      2.4.2 Spine ................. 20
   2.5 Communication between Brain and Spine .... 21
      2.5.1 LeJOS Protocols ........ 21
      2.5.2 Janus Protocol ......... 22
      2.5.3 Addressing ............ 27
   2.6 DirectMode and MacroMode (vertical cut) .... 30
      2.6.1 DirectMode: Rcx and RcxServer .... 31
      2.6.2 MacroMode: Executor and MacroEngine .... 31
   2.7 Complete System Overview ... 38
   2.8 Hardware Architecture (LEGO-Robots) .... 39
      2.8.1 General Requirements .... 39
      2.8.2 LEGO Mechanics .......... 40
3 Implementation

3.1 Macros and Macro Programs ........................................ 41
   3.1.1 An Interpreter for Macro Programs ......................... 41
   3.1.2 Executing Macros ........................................... 43
   3.1.3 Modeling Macro Programs in Java ............................ 43
   3.1.4 Modeling Macros in Java ................................... 45
3.2 Sensitivity ....................................................... 47
3.3 Brain and Spine ............................................... 48
3.4 Communication ............................................... 48
   3.4.1 Hardware Dependent Classes ................................. 48
   3.4.2 Janus Protocol Stack ....................................... 49
   3.4.3 CommManagers .............................................. 50
3.5 DirectMode .................................................... 51
   3.5.1 DirectMode Example ....................................... 54
   3.5.2 Controlling DirectMode .................................... 54
3.6 MacroMode ..................................................... 54
   3.6.1 Executor .................................................. 55
   3.6.2 MacroEngine ............................................... 59
   3.6.3 MacroMode Example ....................................... 61
   3.6.4 Controlling MacroMode .................................... 62
3.7 Robot Prototype ................................................ 62
   3.7.1 Building a physical Robot ................................. 62
   3.7.2 Writing Macros for a new Architecture .................... 62

4 Debugging and Testing ............................................. 63

4.1 Debugging RCX Applications .................................... 63
   4.1.1 LCD and Sound ............................................ 63
   4.1.2 The emu-lejoserun Tool ................................... 64
   4.1.3 The Macrobot Simulator ................................... 65
   4.1.4 JUnit Tests ............................................... 68
   4.1.5 Interactive Tests using DirectMode ....................... 69
4.2 The LeJOS Simulator ............................................ 69
   4.2.1 Design .................................................. 69
   4.2.2 Implementation .......................................... 70
   4.2.3 Usage ................................................... 71
   4.2.4 Limitations and Extensions ................................ 71

5 Challenges and Lessons learned .................................. 73

5.1 iPAQ .......................................................... 73
   5.1.1 Cross-compiling ........................................... 73
   5.1.2 Communication with the RCX ............................... 74
5.2 LeJOS .......................................................... 75
   5.2.1 Patching of existing API .................................. 75
   5.2.2 Integration into the Macrobot Framework ................ 75
   5.2.3 Further Advice ............................................ 76
5.3 Working with limited Memory .................................... 77
   5.3.1 Programming without GC .................................. 77
   5.3.2 Using simple Types ....................................... 78
   5.3.3 Reducing Design .......................................... 78
   5.3.4 New LeJOS Linker ......................................... 78
6 Conclusion

6.1 Results ........................................... 79
6.2 Evaluation ........................................ 80
6.3 Limitations ........................................ 80
6.4 Future Development .............................. 80

A Project Structure ................................. 82

A.1 Directories and Files ......................... 82
A.2 Makefile ........................................ 82
A.3 Scripts .......................................... 83

B Applications ...................................... 84

B.1 Sensor Monitor ................................. 84
B.2 DirectMode Shell ............................... 85
B.3 MacroMode Shell ............................... 87

C Implementing Macros ............................ 89

C.1 Writing Macros for a new Architecture .... 89
C.1.1 Timing ........................................ 89
C.1.2 Slicing ........................................ 90
C.1.3 Architecture-specific Parameters (Tuning) 90
C.1.4 Configuring the Spine ...................... 93
C.2 Adding a new Macro Command .............. 93

D Prototype Robot Model .......................... 95

Bibliography ........................................ 104
Preface

Abstract

This thesis is concerned with the development and implementation of the Macrobot framework, a general framework that supports the deployment of simple robots within an intelligent adaptive environment. Robots are controlled by macro programs that allow to define tasks or behaviors independent of a robot’s architecture. Macrobot was implemented in Java and is intended for use with LEGO MINDSTORMS robots running LeJOS as firmware.

Overview

Chapter one gives a general motivation for our work and introduces the various theoretical and practical concepts that we need for the implementation of the Macrobot framework. Chapter two then conceptually designs a solution for implementing such a framework, and puts the various components from chapter one into a meaningful relation with each other. Chapter three describes the details of the actual implementation. Chapter four gives insight on the problematics of debugging and testing the RCX applications and discusses the implementation of a simulator for LeJOS as a possible solution for the dilemma. Chapter five talks about challenges we encountered and lessons that we’ve learned during the deployment of the system. Chapter six finally gives a conclusion and sketches future work to do.

Appendix A gives an overview and explanation of the project structure. Appendix B introduces various Macrobot applications and describes their usage. Appendix C gives instructions and hints for the implementation and addition of new macros within the framework. Appendix D gives detailed instructions on how to build Carriage, a prototype robot model.

Acknowledgements

My greatest thanks go out to my dearest two flatmates Dominique and Sophie. They have beared my constant state of emergency and the thereby resulting state of chaos and disorder in our apartment with true greatness and comprehension. Through their gentle and cool attitude they have given me a feeling of appreciation and of being part of a small family. I could always count on being cheered up by either one of them if things didn’t work out very well. Extra
thanks go to Dominique for keeping my thoughts flexible by interrupting me all the time during our mutual working sessions in the living room.

I also graciously thank my parents and my sisters for all the loving support and motivation that they constantly gave - and still give - me for everything that I do and ever did since the first day of my life.

With the completion of this thesis my studies at the ETH come to an end, at last. I would thus like to thank all my friends who have accompanied me during the past five years and who have helped in making this part of my life a great and truly unforgettable experience.

Last, but not least, I thank Andrei Popovici who has supervised me for the last four months. He gave me complete freedom of choice about the direction and form that this thesis was going to take and was also very benign towards everything else that I undertook. For his proof reading of the final draft of the report I am very grateful.
Chapter 1

Introduction

In this chapter we first give a motivation for our work and state the exact assignment of this thesis, followed by a detailed example that illustrates a fictional use case. We then define the various components that are required for the Macrobot framework and introduce both theoretical and practical concepts.

1.1 Motivation

Application awareness is the capability of an operating system to adapt applications under its hood, as opposed to context awareness, which defines the capability of a system to adapt with its own forces to the context it runs within. So far, application awareness has been studied with respect to user mobility scenarios. Much less work has been devoted to the question of how to use application awareness for enterprise settings. An enterprise setting is characterized by a larger number of independent and possibly "intelligent" devices such as robots and services. Given device mobility, services and robots must both be adapted to their current working environment.

A possible scenario for demonstrating application awareness could be that of a mobile discovery robot, provided with some intelligence (e.g., capability of recognizing obstacles, mines, walls), but without the knowledge to semantically interpret the significance of each event. For instance, the robot will not be able to distinguish a regular obstacle from a room’s wall. The external application-aware room service will detect the presence of the explorer robot and will record useful information that can be interpreted later and sent to other (non-discovery) robots entering the room.

1.1.1 Assignment

The goal of this thesis is to define and implement a system that supports research in the field of application awareness within enterprise settings. On the implementation level this task requires the integration of physical robots built with LEGO MINDSTORMS and equipped with a two-platform controller (RCX (Hitachi 3800) on one side and iPAQ (StrongARM) on the other) into a distributed system which supports adaptivity by means of Prose. Both the application and the service layer (i.e., the resulting framework) of this environment
will be based on Java technologies.

The implementation part of the thesis focusses on the framework; the realization of the Prose and networking layers is planned for subsequent development stages. However, the resulting system should take those two components into consideration in the design stage.

1.1.2 Example

In order to illustrate our vision of robots that are adapted by an intelligent environment surrounding them we would like to start off with a hypothetic example which presents an adaptation case. The presented case introduces most of the various components and concepts that we will need in order to implement a general framework and should give a pretty good idea of what this thesis is about. The described scenario adheres to the definition of an enterprise setting as it has been given at the beginning of the motivating section on page 3.

Setting

A fleet of mobile, task-bound, low level-intelligent robots is operating within a semi-deterministic environment that also comprises of non-mobile, local, context-aware, high-level controlling entities. Those entities can be thought of as "intelligent rooms" which are able to monitor any activity that goes on within their observed area by means of various sensors such as cameras or light beams. The local room intelligence is aware of its own internal architecture and of any changes being performed within its area, be it persistent or only temporary.

In each room we find a local ad-hoc network, formed by robot instances and the local intelligence residing within the room. Robots may join or leave such a network when they enter or exit a room, respectively, but they must not necessarily be aware of this fact or even know that they are nodes within such a network, it may be transparent to them.

The robots do only have a limited intelligence, just enough for them to perform the task(s) that they have been programmed for, e.g., to move from room A (storage) to room B (production hall), fetch something there (a newly produced item), return to room A (storage) and deposit it there (on a conveyor belt or on a shelf). This task is repeated in an endless loop. The robots are equipped with a basic set of sensors, such as bumpers and a camera or a light sensor. They can orientate themselves by following marks on the floor. There is no explicitly programmed communication between some robot and any other device, each robot just follows a hard-coded program that executes the specific task the machine was built for.

For the sake of this concrete example, we assume that the robots are built with the LEGO MINDSTORMS kit. In addition to the RCX controller, which has no wireless networking capabilities of its own, they also carry an iPAQ, which is equipped with a wireless network card. The robot intelligence is implemented in Java and runs partially on the RCX (using LeJOS) and on the iPAQ. The JVM which is running on the iPAQ runs with the Prose extension and is thus capable of adaption by means of accepting aspects that are sent to it via the wireless network, possibly by using Jini.

It is important to note here that the robot intelligence (i.e., the controlling program) was not built with any kind of adaptiveness in mind. Rather it runs
1.1 Motivation

within an environment which is capable of being adapted, but without having a knowledge of that fact. We should see the situation as if two additional layers had been drawn over the ignorant robot controller: an adaptation layer and a networking layer.

Regular scenario without adaptation

Let us now quickly walk through a fictional scenario describing how some robot R normally picks up a newly produced item in room B and returns to room A with it. Figure 1.1 illustrates the process and setting of the whole example.

![Figure 1.1: Example of an enterprise scenario](image)

To accomplish this task the robot leaves room A and thus also leaves the local network formed around the local intelligence in A. It moves down a hallway, following some marks on the floor or navigating by any other means that we can think of. For the time being we just assume that it turns right as soon as it crosses a red line, marking the doorway for room B. The hallway should actually be seen as another room C, with all the characteristics that we have described above. So R becomes a node in C, for a short time, until it leaves the C-network again and becomes part of the B-network as it enters room B. Here it usually goes straight forward and parks on the first of a number of pick-up spots, waiting for a new item to be delivered.

As soon as it has received its load, it turns around and returns to room A where it delivers the cargo and immediately performs the next cycle of its chore.

Temporary scenario change

Let us now assume that room B temporarily stops its production for maintenance reasons. Our robot can no longer carry out its duty for obvious reasons. We cannot assign it to another task, since it was built for this work only, so we probably have no other choice than shutting it down during the time that room B undergoes maintenance.
This is a rather unpleasant solution, since we are waisting resources. We particularly don’t like it, because we know that there is another room B’, just a bit further down the hallway, which is similar to B, and which could serve our robot, while B is out of service. But R does not know how to get there, and we do not have the option of changing its program on the fly.

However, since the robot has adaptive capabilities, we can do what we would like to do: we can have R pick up its load in B’ rather than in B for a particular period of time.

**Exceptional scenario with adaptation**

How can we adapt our robot so that it is capable of dealing with the new circumstances? The "intelligent room" will perform this adaptation transparently for the robot, i.e., the machine won’t even notice that it is being adapted to the new situation. How exactly this is done is illustrated best by walking through the same example again.

As before, R leaves A and enters the hallway C. C notices R and is aware of the fact that the robot should be adapted and guided to room B’ instead of room B. It thus sends an extension to R which changes the robot’s behavior in such a way that it now actually waits until it has passed a second red line before turning right to correctly enter B’ instead of B. As soon as the robot leaves the C-network, the extra extension is withdrawn.

R now stands in the entrance of B’. The two production rooms are identical in shape so there is not much that needs to be adapted here. However, R cannot park in its usual spot, because there is already another robot waiting to be served. B’ is aware of this problem and sends another extension to R making the robot park in the spot just next to the one it is used to. Another extension is sent which corrects the robot’s movements in order to find the exit of B’ again, after it has received its cargo. Back in the hallway, the extensions issued by B’ are removed again. At this point, our robot still "thinks" that it has just left B as usual. Yet another extension sent by the hallway intelligence takes care that it finds the way back to A where it executes its normal program.

### 1.2 Robots

![Feedback loop of a robot](image)

A robot is a device that responds to sensory input. Robots are normally driven by a feedback-loop like the one depicted in figure 1.2. They are typically designed to execute one or more tasks repeatedly. Tasks may be issued by a human operator, but most robotic systems are controlled by computer.

*Autonomous robots* act as stand-alone systems and are controlled by their own computer, they are often called *agents*. An agent is endowed with some artificial intelligence so that it can react to different situations that it may
1.3 Adaptation

The amount of intelligence is not necessarily required to be high; many agents are just controlled by simple state machines.

An agent, or a robot in general, must not necessarily be a physical device. It may also be realized virtually as a software system. However, this thesis only deals with physically implemented machines.

Robotics, i.e., the field of computer science and engineering that is concerned with the creation of robots, is understood to be a branch of artificial intelligence.

1.3 Adaptation

The term adaptation is defined as "the act of changing so as to fit or become suitable". For software systems, adaptation usually implies the automatic integration of an application into an environment of a different nature. The spectrum of adaptation is delimited by two extreme forms: on one end, it is the responsibility of the application to adapt itself to the environment (laissez-faire adaption), on the other end, solely the environment is concerned about adapting the applications that run within it (application-transparent adaption). Thus software can be both actor and patient in an adaptation process.

The range of possibilities that lies between laissez-faire and application-transparent adaption is called application-aware adaption.

Application-awareness has been studied in the Coda and Odyssey experiments [4], with respect to mobile information access.

For the implementation of the Macrobot framework we are primarily interested in application-transparent adaption strategies. Robots should be adapted to their environment without being aware of it. All of the adaptation is performed by the intelligent environment (i.e., by "intelligent rooms").

1.4 Lego Mindstorms

With Lego Mindstorms, the danish toy producer Lego has made a cheap and robust, yet very versatile robotics construction kit available on the market.

1.4.1 Parts

A basic Mindstorms kit consists of about 700 Lego Technics parts, two motors, three sensors (2x touch, 1x light) and a programmable micro controller, called RCX, which lies at the heart of every Mindstorms construction.

RCX The RCX contains a Hitachi H8300 micro controller and 32 kB RAM on board. The brick exposes six ports on its upper surface that can be used to read or control sensors and motors (three each), respectively. Communication with a PC for uploading programs and for exchanging information during operation is performed by means of an infrared connection. Three programmable buttons and a five-digit LCD display complete the interface that the RCX can use to interact with its environment. The RCX has been almost completely reverse engineered, technical details about nearly every aspect of the RCX’s internals are available from [17].
Sensors  Sensors give a robot the ability to sample data from its surrounding world and to initiate an appropriate action according to the interpretation of that data. Light and touch sensors are part of the basic MINDSTORMS kit, but there exist many more such as temperature, rotation or proximity sensors, for example. It is not very difficult to build an own device either, the internet produces dozens of instructions on this topic. All sensors produce raw value readings in the range between 0 and 1023, inclusive.

Motors  Motors introduce the chance of both mobility and physical interaction to a robot. The RCX can control at most three motors independently. In combination with gears, wheels and hydraulics (available from the regular LEGO TECHNICS products), motors allow us to build complex drives or robotic arms, for example.

IR-Tower  All communication with the RCX is performed over an infrared link. For this purpose the workstation that runs the RCX programming environment must be equipped with a so-called IR-tower. This device acts as an infrared port for the PC and is available both as an USB or a serial version.

1.4.2 Prototyping

The LEGO system provides an excellent way for building robot prototypes. Through its three funding concepts, reusability, connectivity and modularity, LEGO becomes a very powerful building tool. The same brick can be used over and over in different contexts without being damaged, no matter how many times it is used. One doesn’t need glue or screws to assemble or dismantle a LEGO model. Instead all of the pieces easily snap to each other and stay firmly in place until they are taken apart again. We talk of modularity, because the individual LEGO bricks connect to one another in exactly predefined locations. Altough this may seem like a disadvantage at first, it is not, because it allows for very precise positioning of parts.

Due to the features just stated, assembly of a new robot model is very fast, compared with other alternatives. If a new architecture proves to be ill-conceived, it is quickly taken apart again. The LEGO system offers hundreds of different parts and thus it is rather unlikely that we cannot build what we have in mind (within reasonable pretensions of course).

Most certainly this does not mean that it is simple in principle to build sophisticated robots. But LEGO makes it very easy to explore new ideas quickly and to optimize an existing architecture by constantly improving it. Together with the special MINDSTORMS components that we’ve described in the previous section, LEGO really is an an ideal prototyping kit for robotics applications.

1.5 Programming Environment

To provide means for steering and controlling of the MINDSTORMS robots that will be deployed within the Macrobot framework is a central assignment of this thesis. How will we implement programs that run on the RCX and what programming environment do we employ for this task?
1.5 Programming Environment

1.5.1 Rcx Code

The first thing that comes to mind is to use RCX Code, the MINDSTORMS SDK, which is part of every robotics kit. Rcx Code is a proprietary graphical software developer’s kit that is targeted at an audience not experienced in programming. Programs written with RCX Code are compiled into a sequence of opcodes and are then uploaded to the RCX via infrared. There are several disadvantages of that system though, the most significant ones being the following: unsatisfactory level of access to the internals of the RCX, no support of communication with the PC (upload only) and no way to program multi-threaded applications. Last but not least, one soon gets tired of assembling programs from graphical “command bricks” using the mouse.

1.5.2 Alternative RCX Programming

Within several weeks of the first release of MINDSTORMS, the hardware internals of the RCX were reverse engineered and made available publicly to the interested. A few months later, the open source community had produced a series of alternative ways to program the RCX, sometimes along with alternative firmwares. The most important of those are:

BrickOS [8], a C-based firmware which allows to write robot programs in standard C, using the standard GNU compiler tools. BrickOS gives complete low-level access to all of the RCX’s internals. It is used mainly by system programmers who desire to take advantage of the last bit on the RCX.

NQC [9], a high level language with a C-like syntax. Comes with a compiler to produce opcode programs for the original LEGO firmware. NQC is the best established alternative programming system for the RCX. It is powerful yet easy to learn and not as complex as the original C-language.

pbForth [10], a FORTH dialect. FORTH has a long tradition in robotics research and is used in many ”real world” applications. pbForth (the pb stands for ”programmable brick”) comes with an own interpreter for the compiled programs which runs on the RCX.

LeJOS [11], a small Java Virtual Machine (JVM) implementation that runs on the RCX. This firmware gives users the chance to write control programs for robots in pure Java and comes with a fairly large API. LeJOS also includes a communications package that makes it very easy to establish a stream-based communication between PC and RCX.

Each of those high-level programming language implementations gives access to all of the RCX hardware and allows to write complex programs for the Hitachi 8300 micro controller. Still, for the Macrobot project only one of them is a suitable choice, namely LeJOS.

The main rationale behind this decision is that we strive to use Java, for practical reasons. We wish to provide a homogenous environment for the programming and deployment of our robots as well as for the implementation of the Macrobot framework. Since the application layer, the adaptation layer (Prose) and the networking layer (Jini) are already based on Java technologies, it would be rather unwise to choose an ”incompatible” language for the realization of the robot controller.
Other Java Solutions

By choosing LeJOS for the implementation of the RCX-based framework parts we decided to install an alternative firmware on the RCX.

However, if it were just for the sake of using Java, we could have chosen from several other alternatives that also allow to control the RCX, but without uploading a new firmware. There exist several Java libraries that allow to program the RCX remotely:

**RcxDirect API** [12] is a very simple package for remotely controlling and reading the RCX hardware components. A server program must be uploaded to the RCX to handle the requests.

**RcxPort API** [13] is a package that allows for construction of programs (consisting of commands and subroutines) from opcodes and sending them to the RCX. This API is working on a rather low level and does not need a receiver program on the RCX.

**RcxJava API** [14] is a high level API which lets the programmer write Java applications for the RCX on the PC. It offers abstractions for the most common RCX hardware. Programs are executed on the PC and any commands related to the RCX hardware are translated into opcodes and transmitted to the RCX, resulting in a communications delay between command issue and command execution.

We list those only for the sake of completeness. For obvious reasons, none of the above stands a chance against a full-blown, Java-based operation system for the RCX, like the one that we get with LeJOS.

1.5.3 LeJOS

LeJOS was originally conceived as *TinyVM* [15] by Jose Solarzano who was challenged by the task of implementing a Java virtual machine for the RCX platform. The resulting interpreter had a size of about 10 kB which left about 18 kB RAM for Java programs and offered the following features (among others): Java on the RCX, preemptive threads, arrays, recursion, synchronization and exceptions. After the stable release of TinyVM the effort of completing and extending TinyVM was continued by other open source developers as the LeJOS project. The interpreter grew in size up to about 17 kB, almost inverting the ratio of program memory vs. firmware size. However, in addition to everything already offered by the original TinyVM, LeJOS adds the following features: floating point operation, string constants, casting of longs to integers and vice versa, reference marking in the stack (making it feasible to implement garbage collection), multi-program downloading, `java.lang.Math` implementation and more APIs.

Because of its small size, LeJOS also lacks some of the comfort that a regular Java VM offers to the programmer. Top of the list is the missing garbage collection, but other features such as the `switch` statement or the `instanceof` operation are neither available. Yet all of the missing attributes only reduce the coding comfort and can easily be dealt with or programmed around if one is aware of them.
1.6 Aspect Oriented Programming

API
Due to the size constraints and lack of man power, a lot of the standard Java
API is not available. However, the current API covers the most important
classes from java.lang and java.util, as well some classes from java.net
and java.io for communication purposes. The mindstorms specific libraries,
bundled in the josx.platform.rcx package offer abstractions for nearly every
bit of hardware that can be found on or steered with the RCX brick. Using
the josx.rcxcomm package one can easily establish a communication between
PC and and RCX or between two or more RCX’s, based on various reliable or
non-reliable protocols. Additionally, there are several classes providing classic
patterns for the implementation of robotic intelligence and behavior. More
libraries (both standard Java and MINDSTORMS specific) constantly evolve from
the open source community, as they are implemented whenever the need for
them arises.

Most of the josx.rcxcomm classes have also been implemented for the PC
platform. Applications running on the PC that wish to interface or communicate
with the RCX must include those PC-specific classes on their classpath instead
of the leJOS classes.

LeJOS 2.0 supports Windows, Unix/Linux and Mac platforms. USB towers
are supported under Windows and under MacOS, development is on the way
for Unix/Linux systems. Serial towers can be used without problems on all
platforms.

Tools
The LeJOS distribution comprises of a linker that produces a binary file from
class files and a loader that performs the upload of this binary file to the RCX.
An uploaded binary file is copied verbatim into memory on the H8300 and
executed as soon as the user presses the ON button on the RCX brick. LeJOS
does not include an own compiler, instead it uses any existing Java compiler
(e.g., javac or jikes) together with a non-standard bootclasspath pointing to its
own libraries.

Some extra tools exist to aid the process of debugging of LeJOS programs,
a task that can be really difficult. They are described in section 4.1.2.

1.6 Aspect Oriented Programming

Aspect Oriented Programming (AOP) is a style of programming that attempts
to abstract out features common to many parts of the code beyond simple
functional modules and thereby improve the quality of software.

Many software systems have properties that don’t necessarily align with
the system’s functional components, such as failure handling, persistence, communica-
tion, replication, coordination of memory management or real-time con-
straints. Those properties can not be isolated into individual modules by current
OOP methods, because they tend to cut across groups of logical modules.

While they can be thought about and analyzed relatively separately from

\footnote{The next release of LeJOS (2.1) is expected to support the USB tower under Linux.}
the basic functionality, programming them (with current OOP languages) usually results in a tangled mess of instructions for different purposes.

AOP offers a solution to this "tangling" problem by giving a way to express such aspects of concern in a separate and natural form. The separate descriptions can then be automatically combined into an executable form together with regular code.

It is important not to see Aspect Oriented Programming as a completely new form of programming but rather as an extension to the existing OOP paradigm. This is also reflected in the fact that all currently existing implementations of AOP languages work in collaboration with an established OOP language.

[5] and [18] give more detailed information on the subject of AOP.

1.6.1 Prose

Prose [6] [7] [19] is an AOP extension for Java. Rather than being implemented as an own language it allows for the definition of aspects directly within Java, using the constructs and components of the Prose libraries. Aspects are defined as Java classes and then compiled into ordinary class files. The speciality of Prose is that the system - in contrast to most other AOP implementations - supports the insertion and removal of compiled aspects at run-time rather than at static compile time. Using this mechanism we can change Java programs while they are running, a fact which makes Prose the perfect vehicle for realizing adaptive systems.

Prose can, by design, be used very easily in collaboration with Jini, a framework for "spontaneous networking", developed by SUN Microsystems [20]. The additional Jini layer on top of Prose gives users the capability of inserting and removing aspects into/from running applications over a network. This feature makes Prose particularly suited for being used together with the Macrobot framework in order to realize a scenario like the one described in the introductory example at the beginning of this chapter.

The inclusion of Prose into the Macrobot system will not be a part of this diploma thesis. But runtime adaptation within the Macrobot framework will be performed on the basis of Prose, at a later stage of development. Thus design and the implementation of the system at hand should, to some degree, anticipate the employment of Prose at a subsequent point of time.
Chapter 2

Design

This chapter illustrates the conception and the design of the Macrobot framework.

After the analysis of the requirements, we introduce the concept of macros and sensitivity. The upfollowing sections then perform a horizontal (Brain-Spine) and a vertical (DirectMode-MacroMode) cut through the architecture of the Macrobot system along with a discussion of the communication protocol which is employed between the Brain and the Spine processes. A complete conceptional overview of the framework concludes the software design part. The last section eventually discusses various important design aspects that should be taken into consideration when building physical robots with LEGO MINDSTORMS.

2.1 Requirements Analysis

The main goal of this thesis is to build a framework that allows to implement, test and run scenarios similar to the one described by the lead-in example in chapter 1. In that very chapter we have already decided about the hardware platforms, the programming language and the concepts that we are going to use in order to achieve this goal. However, many more decisions must be made and many more requirements must be met. The following points define the most important ones of them.

Components Several different existing components, both in hard- and software, must be integrated with each other. We introduced, or at least mentioned, all of those components in chapter 1. In no particular order they are:

- various LEGO MINDSTORMS hardware components (e.g. IR-Tower, Sensors, Motors, RCX) which are typically part of a physical robot implementation
- an iPAQ, equipped with a wireless LAN card, supporting the RCX with computation power and binding the robot into the network
- clients (typically human), who desire to upload new tasks or behaviors to any robots which are part of the local wireless ad-hoc network
• non-human (i.e. automated) clients, also connected over the wireless network, desiring to adapt robotic behavior by means of sending Prose extensions

• LeJOS, a JVM for the RCX programmable brick

• RCX-IR, a native library that implements communication with the RCX using the built-in IR-port of a laptop or an iPAQ and thus rendering the need of a tower obsolete. The RCX-IR project is still in a beta phase (as of December 2002), however, its integration would be a very "nice to have" for our system.

• Prose, a dynamic AOP extension for Java, that can be used to build adaptive systems or environments

Adaption  Adaption must be supported. Prose adapts software by inserting extensions at runtime before or after method code is executed. Our system should thus provide suitable insertion points (i.e. crosscuts) for extensions that perform adaption. In other words: the Macrobot design should anticipate adaptation by Prose.

Generality and Abstraction  Since the framework will be used for research purposes it should be as general as possible. We don’t know yet what kind of robots will become part of the framework, hence it must allow the definition of behaviors and tasks that are (at least partially) independent of the hardware architecture of any future robot models that will be deployed. This fact demands an abstraction from the concrete robot hardware. However, we assume that research will mainly work with mobile robots and that adaptation will mainly be performed with respect to displacement. Therefore we require abstraction with respect to steering commands only.

Usability  It should be easy to integrate new robots into the framework. Programming and re-programming of robots should be a simple task for both human and non-human clients. Overall, support for rapid prototyping is a headrequirement. If desired, access to the low-level API of a robot’s hardware should be possible (i.e. it should not be complicated to bypass the abstraction layer requested in the previous paragraph).

Limited Resources  The H8300 microcontroller on the RCX has only 32 kB of memory in total and limited calculation power. With LeJOS installed, the amount of memory available for user programs shrinks to about 11 kB in total. It is evident that those limitations must be taken into consideration when designing an application like the one at hand as they signify a bottleneck for the whole framework.

Application Load  The 11 kB of memory that are normally available for user programs on the RCX usually suffice for implementing rather sophisticated robotic behavior for a specific application. Building a general solution or one that supports quick development and integration of various heterogeneous solutions or components will always result in a larger size. The overhead is caused
by the more general design and the openness of the architecture. Running Prose and other Java extensions (e.g. networking components) side by side on top of such an architecture, does certainly not reduce our need for memory and computation power. All together, the system we are about to build clearly exceeds the limitations of the RCX hardware. It therefore seems a logical decision to put the main load of the application (and thus the main "intelligence" of the robot controlling software) on the iPAQ and to use the RCX as a hardware-controlling backend only.

**Non-Requirements** Constructing steerable and maneuverable robots that are capable of performing one or more specific tasks in a satisfying manner is a non-trivial task which should not be underestimated and which requires a good working knowledge of robotics in general and mechanics in particular. Guidance or solutions for building a "good" robot architecture is regarded as an engineering task of its own and will not be part of this thesis.

The development of adaptive software is not part of this project either, the *Macrobot* framework will merely provide a basis for the deployment of such applications. Similarly, networking capability (e.g. for RMI) is anticipated and part of the design but not a major goal for implementation.

## 2.2 Macros

As required in the previous section, *Macrobot* is to be a general steering framework for robots and should support abstract steering commands in order to become independent from the actual hardware. If we have three different robot architectures, say a steering wheel, a differential drive and a caterpillar architecture, we would like to write programs for all of them that look alike. We are not interested in the various detail commands that are issued to drive the individual hardware (motors for example), making the robot turn by 90 degrees. Rather we would like to be able to issue a simple `turnLeft(90)` command and compose our whole programs of such high-level commands only.

Macros provide such an abstraction. We hereby define a macro to be an abstract command specification which is implemented differently for each robot architecture. Calling a macro will cause the execution of a series of architecture-specific commands, which eventually achieve the goal specified by the abstract macro command. The set of all macros available within the *Macrobot* framework forms a general driver interface which is implemented separately for every specific robot architecture. The actual hardware details are thus rendered transparent by the framework and become subordinate for the writer of macro programs, and, more important even, for any entity wishing to adapt or influence a robot’s behavior externally.

Table 2.1 presents the minimum of commands required to maneuver a mobile robot with two degrees of freedom to any desired position. Macros for all of those commands (or similar ones) should be implemented as a minimum requirement by any new robots employed within the *Macrobot* platform. Please note that this list is only a minimal one and can be extended with arbitrarily many more commands within an actual system.

The case may arise that a robot is simply not capable of performing a certain macro with the desired degree of accuracy, due to physical constraints
Table 2.1: Minimum list of macros

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{forward(distance)}</td>
<td>Makes a robot move forward by a given \textit{distance} in centimeters.</td>
</tr>
<tr>
<td>\texttt{backward(distance)}</td>
<td>Makes a robot move backward by a given \textit{distance}.</td>
</tr>
<tr>
<td>\texttt{turnLeft(angle)}</td>
<td>Makes the robot turn in place, counterclockwise by \textit{angle} degrees.</td>
</tr>
<tr>
<td>\texttt{turnRight(distance)}</td>
<td>Makes the robot turn in place, clockwise.</td>
</tr>
</tbody>
</table>

raised by its architecture. A robot without a differential drive, for example, is physically not capable of turning in place. If the robot is facing north, a \texttt{turnLeft(90)} command will always result in a displacement of the robot’s center point to the northwest, even though the machine will correctly be facing westwards when the turn has completed. In other words, such an architecture (e.g., a regular car) can only turn on a radius of a certain size. This, however, shall not concern us here, as exact navigation is yet another non-trivial problem in robotics. The errors which are resulting from inaccuracies like the one just described, are for most cases insignificant for the type of applications and problems we wish to analyze and investigate\(^1\).

2.2.1 Macro Programs

Taken in isolation, macros are limited as they just define abstract movements of a robot. Their full power does not arise until they are used in combination with each other, forming a sequence of abstract commands, i.e. a \textit{macro program}. Such a program should ideally be a truly architecture-independent algorithm which positions and drives robots of different (even unknown) architectures identically.

Macro programs can implement a whole basic behavior of a robot (e.g. exploring a room), a specific task (e.g. traveling from room A to room B) or just a single, isolated action (e.g. driving around an obstacle which is in the way), which may be part of a task.

The \textit{Macrobot} package allows to define any of the three classes of programs just described. Each of them has a different precedence with respect to the others and is executed accordingly (the exact differences will be explained in section 2.6.2).

\(^1\)Thinking of it, we might even declare the physical disability of a robot to perform a sequence of macro commands with the desired degree of accuracy for a new problem that we’d like and actually can solve by using adaptation. All we have to do is to shorten the distance that a robot travels straight forward just before and possibly after the execution of a turn command, taking the radius and relative displacement that are resulting from the turn into account.
2.3 Sensitivity

As we know from section 1.2, a robot is somewhat helpless and of limited use if it cannot receive or process feedback from the environment it is acting within. This is particularly true for agents living in a non-deterministic environment. Here, something unexpected may happen at any time, in which case a robot might have to take appropriate actions depending on the data it receives from its sensors and depending on the goal it pursues.

But even if a robot lives in a more deterministic environment, e.g. in a static world which is not bound to change over time, feedback is vital to most applications. Feedback allows a robot to continuously check its surroundings and to update its internal state accordingly, i.e. to synchronize itself with the outer world that it lives in.

Most agents are not all-knowing with respect to the world they act in, even if it proves to be of static and deterministic nature. Often a robot has not enough computing or memory power to save or model the complete universe surrounding it. Also we must realize that even a static universe is often simply too complex or too large to be completely understood or be modeled. In such cases, a robot may only be aware of particular characteristics of its surroundings and mainly take action with respect to those characteristics. The features of interest are typically detected by means of sensor-readings, which can be of many various types, ranging from, for example, simple temperature measuring up to the digital processing of a camera image.

Naturally, all robots deployed within the Macrobot framework should also have the opportunity to process sensor inputs. For this reason, the framework supports a feature which we call sensitivity. It means, that a program can execute a macro or a series of macros sensitive with respect to one or more of a maximum of three sensors. The programmer can set certain range or threshold value which should be observed on a specific sensor. If the values of the sensor enter the sensitive value range or go across the threshold, the program immediately stops all of the current activity and enters some handling procedure, which will perform some appropriate action and thereafter resume the robot’s activity. The procedure we’ve just described is, in many ways, very similar to interrupt handling in an operation system. As a matter of fact, macro programs are really designed to be executable entities which are run on a virtual machine and which - by means of sensitivity commands - actually set or remove both a filter and a handler for specific interrupt requests which are sent by sensors mounted on the robot.

2.3.1 Handlers and Parameters

The basic LEGO MINDSTORMS set includes two types of sensors, namely touch and light sensors, but many more are available. All of them read values in the range of 0..1023 that are internally translated into percent values by the Macrobot system.

A macro program declares one or more portions of its code to be sensitive with respect to a specific sensor. Those regions, we call them sensitivity blocks, may be intersecting or disjoint. An example of code with sensitivity blocks is given by figure 2.1. Each sensitivity block is assigned a handler procedure and a filter, the latter being a set of parameters defining exactly when and for which
values on the targeted sensor the associated handler will be called.

The parameters define mode, type and threshold and the ID of the targeted sensor for each sensitivity block. The meaning and possible values for each of those attributes are the following:

**Mode** defines whether the given threshold should be interpreted relative or absolute. The former defines threshold to be a change over time, i.e. a change between two sensor readings, whereas the latter defines it to be an absolute sensor value.

**Type** is one of upper bound, lower bound, upper bound pulse, lower bound pulse or edge and defines when exactly a sensitivity event (i.e. interrupt) should be thrown, that is, under which conditions the provided handler routine should be called. If type is set to upper bound, then a sensor event is thrown every time a value lower than the given threshold is read from the target sensor. If upper bound pulse is specified, then an interrupt will occur every first time that a value lower than the given threshold is read. Lower bound and lower bound pulse are interpreted accordingly with values greater than the given threshold. The edge value, at last, invokes the associated handler every time when the value function on the target sensor intersects with the threshold value, i.e. it combines upper pulse and lower pulse.

Figure 2.2 graphically shows the differences in event generation, depending on the sensitivity type used.

**Threshold** is simply a value between 0 and 100 (inclusive) which will be interpreted according to the mode setting.

**ID** is the code of the sensor that should be observed. This can be a relative value, such as front bumper or bottom light, chosen from a list, or an absolute specification, i.e. one of S1, S2 or S3. If the sensor does not exist for the robot which is steered by the current program, then an exception event will simply never occur.
2.4 Brain and Spine (horizontal cut)

The application load requirement from section 2.1 strongly suggests a horizontal subdivision of the framework’s architecture. As a consequence the structure of the system was split into a Brain and a Spine package. The Brain’s components include everything that will run on the iPAQ on a regular JVM; the Spine’s components are the ones executing under LeJOS on the RCX. The general analogy we try to establish here, is that the main intelligence executes on the iPAQ which thereby forms the ”brain” of the robot, whereas the RCX forms the ”spine” or, more precisely, the ”spinal marrow” which merely executes reflexes that are initiated by the brain.

Brain and Spine form two independently running applications that run concurrently on different platforms, i.e., on an iPAQ (or PC) and on the RCX, respectively. They communicate with each other over an infrared link. A separate communication package (see section 2.5), based on some low-level com-
munication components which are part of the LeJOS API, implements an own messaging-protocol between the two platforms.

2.4.1 Brain

The Brain application comprises roughly of the following components:

Executor The Executor is the virtual machine which runs the macro programs that are sent to a Macrobot by a client. Section 2.6.2 gives more insight on how the Executor works.

Rcx This abstract component defines and exports a low-level API offering direct access to the most important functions of the various MINDSTORMS hardware components which are either attached to or part of the RCX programmable brick. This is explained in further detail in section 2.6.1.

BrainController The BrainController exports the "Brain API". Any clients that wish to interact with a Macrobot instance do so through the methods which are exported by the controller. This component also creates and initializes all of the Brain’s components at instantiation time.

Furthermore we attribute all of the Execution Model components (see section 2.6.2) to the Brain package.

2.4.2 Spine

The Spine system contains the following parts:

Macros All of the architecture-specific macro implementations which are invoked by various macro programs running in the Brain.

MacroEngine The MacroEngine fetches, loads and executes the selected macros by request of the Brain. It must provide functionality to control the execution of macros, i.e., to stop, finish or continue them at any point of execution.

SensorWatch A poller thread which continuously reads sensor values and which produces sensitivity events when appropriate.

RcxServer A thread that serves requests sent through the low-level API exported by the brain’s Rcx component.

SpineController The SpineController creates and initializes all of the Spine’s components at start up time and provides more general Spine services which can be used for testing or configuration purposes.

All of those components will eventually be compiled and linked into a binary executable before they are uploaded to the RCX. Compilation, linking and upload are performed by using the various LeJOS tools.
2.5 Communication between Brain and Spine

The Brain and Spine applications are not only logically, but also physically separated. Albeit their forming of two separate processes they frequently need to exchange information. Thus the need for a stable communication channel between the two packages immediately arises. As both Brain and Spine run concurrently and both may produce asynchronous messages for their peer we are basically in need of a duplex communication channel. But the IR link between iPAQ and RCX is of half-duplex nature only. This is not really a problem though, since duplex communication can be simulated on top of a half duplex physical medium using an appropriate protocol.

2.5.1 LeJOS Protocols

The LeJOS API contains an own package, called rcxcomm, which implements several communication protocols. Table 2.2 enlists all of them together with their major features.

<table>
<thead>
<tr>
<th>Protocol Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F7</td>
<td>The F7-protocol is the protocol employed by the original LEGO firmware on the RCX to communicate with the PC. F7 ensures packet integrity by using checksums and bytewise complements, but the protocol is not reliable.</td>
</tr>
<tr>
<td>LLC</td>
<td>LLC is a proprietary LeJOS protocol. It is based on the LLC (“Low Level Comms”) class that can be found in the rcxcomm package and which bypasses the RCX’s ROM routines for serial communication. The LLC protocol allows for sending raw byte packets and provides neither reliability nor integrity.</td>
</tr>
<tr>
<td>LNP</td>
<td>The Lego Network Protocol has originally been developed for BrickOS, another alternative RCX firmware, and was thereafter ported to LeJOS. It is an addressing network protocol that supports addressing of up to 15 stations plus broadcast, and which uses a circular checksum for integrity. It is not reliable.</td>
</tr>
</tbody>
</table>

Table 2.2: The LeJOS protocols

An analysis of the mentioned protocols was performed with regard to a possible usage for resolving the communication needs within the Macrobot system.

Concerning speed, our observation was that F7 proves to be a relatively slow protocol, whereas LLC and LNP are quite fast. LeJOS provides complete protocol stacks for all of its protocols, some with additional layers that implement reliability. LeJOS even supports the usage of streams on top of the mentioned protocols.

Unfortunately, none of the LeJOS protocols supports duplex communication, allowing both parties to send at the same time without scrambling each
others packets. All of support inherently synchronized communication only\textsuperscript{2}. Implementing an additional synchronization layer on top of a reliable communication stack or on top of streams would achieve the desired duplex functionality, however, it seems to be conceptionally wrong to place synchronization there. Apart from being an unfavorable design decision, this solution would be rather ineffective and memory consuming due to the usage of too many components and the replication of already existing functionality from other layers.

As a result of our analysis we have rejected all of the readily available protocols for our purposes. Since this left us without any suitable alternatives, we had to develop our own duplex protocol. The next section describes in detail the specification and design of the \textit{Janus Protocol}, which has been the result of such development.

\subsection*{2.5.2 Janus Protocol}

The existing protocol stacks (LNP, LLC and F7) only support half-duplex communication with error-recovery. There is no synchronization between PC and RCX and thus, if both ends try to send at a high rate, collisions will occur endlessly and both parties will end up deadlocked. When designing a duplex protocol, we must first provide an additional synchronization layer that avoids such a deadlock scenario.

Such a layer could be implemented by introducing a randomized waiting period before the next retransmission (as, for example, \textit{Ethernet} does it) once a collision has been detected. But the waiting period would have to be rather long and cannot guarantee collision-free retransmission at the next attempt. This solution is feasible on a carrier with many senders and many receivers, where explicit synchronization among all parties would demand an overly complex protocol. Such protocols exist as \textit{master-slave protocols} (where slaves are granted permission to transmit by a master station in the network), \textit{token protocols} (the station which owns the token is allowed to send and thereafter hands the token on to the next station) or as \textit{time- or frequency splitting systems} with an external clocking line for synchronization (which is obviously not a practicable solution for our environment).

But since we know that our communication channel has exactly two endpoints, the realization of a synchronization protocol is rather straightforward. We base the design of such a protocol on a defined start state for both endpoints (one being the master, the other being the slave) and on single-bit sequence numbers.

The master initiates communication, by sending data. The slave waits for data in the beginning and can only send data by piggybacking it onto \textit{ack} messages for any received data. If no data is to be sent, either side just sends plain \textit{ack}s, keeping the two parties in sync. Sequence numbers are incremented by the master only; the slave simply bounces any received sequence numbers back at the master.

Because the sequence numbers are a central feature of this communication, we must make sure that they are transmitted correctly. Any duplex communi-

\textsuperscript{2}E.g. for settings in which one party always expects the other end to send data. No asynchronous sending is performed or, if it is, then with arbitrarily long time intervals between moments of sending data.
Both processes regulate access to their send and receive buffers with boolean variables, the send buffer stays locked until it is absolutely clear that the last packet was received by the peer station (i.e. written to the receive buffer at the other end).

Because of its two-faced nature (where one of the communication endpoints must play the master, the other the slave, but it does not matter which one is which) we named the protocol "Janus". Together with a link layer and an integrity layer underneath, it forms a minimal communication stack ("Janus protocol stack") that supports duplex communication.

The pseudo-code snippets following below describe an algorithm that implements the Janus protocol as we’ve just sketched it. Remember that all sending and receiving is is done via a lower integrity handler.

Master

The master endpoint initiates communication by sending an ack with sequence number 0. If the master times out during its receive period, we just resend the last data or ack with the same sequence number.

If we receive a packet carrying the current sequence number, it can be safely assumed that the last send has succeeded and that we are in sync. If the returned sequence number is wrong (i.e. different than the current one), our timeout value is too short\(^3\). In that case, we ignore the received packet, emit a warning and use an appropriate algorithm to increase the timeout period by some value. After pausing for as long as the newly adjusted timeframe, any packets which arrived while waiting are disregarded and we start all over again. This measure hopefully clears the channel from any duplicated packets and brings master and slave into sync again (if the timeout value is still too short, it will be adjusted further in the next run).

Please note what happens in case of a successful transmission (i.e. current seq number comes back from slave), where the slave sent some data but some formerly sent data hasn’t been fetched by the client of the master yet. This would actually result in a buffer overflow (we can’t put the newly received data anywhere). If this case occurs, the master just treats it, as if the last transmission had been a failure: it resends the last packet with the same sequence number, which will cause the slave to resend its data again, giving the master’s client another time cycle to fetch the old data.

For this protocol it is only important, whether the new sequence number is different from the last one (or identical). Therefore it is sufficient to increment modulo 2 and actually flip between 0 and 1.

---

\(^3\)If the master sends a packet to the slave, but times out before the answer arrives, it will resend the same message. If now, shortly after resending, the answer of the slave arrives, it will take it for the answer to its second send and increment the sequence number. If the real answer to the second packet arrives now, we have a sequence number mismatch.

Actually this scenario is rather unlikely to happen, as the two parties will rather deadlock, because we only have a half-duplex medium and the resent message of the master will collide with the answer of the slave. A sophisticated protocol would increase the timeout period if the number of subsequent timeouts is too large.
// set up
byte[] sendBuf = empty;
byte[] recvBuf = empty;
byte[] tempBuf = empty;

int currSeq = 0; // master starts with 0
byte[] outBuf = ack(currSeq); // start by sending an ack

int seq;
int nofTimeouts = 0;
boolean lastSendTimedOut = false;

// master loop
while (true) {
    // master starts with sending
    send outBuf;
    // receive (with timeout)
    tempBuf = receive data;
    seq = receive seq;
    if TIMEOUT {
        nofTimeouts++;
        if (nofTimeouts > MAX_SUBSEQUENT_TIMEOUTS) {
            // had too many timeouts, adjust timeout period
            adjust timeout;
            nofTimeouts = 0;
        }
        lastSendTimedOut = true;
        goto begin;
    }
    lastSendTimedOut = false;

    if (seq == currSeq) { // last send was successful
        if (tempBuf != ack && recvBuf != empty) {
            // buffer full! request again
            goto begin;
        }
        currSeq++;
        if (recvBuf == empty && tempBuf != ack) {
            recvBuf = tempBuf;
        }
        if (sendBuf != empty) {
            outBuf = sendBuf(currSeq);
        }
    }
}
clear sendBuf; // release for new data
}
else {
    outBuf = ack(currSeq);
}
}
else {
    // time out value is too short, adjust
    adjust timeout;
    wait until timeout;
    drop all packets that arrived while waiting;
}
}

Slave

The slave expects a sequence number different from the last one received. If this is the case, the last data packet sent was received by the master, who then incremented the sequence number. We can thus safely assume that our last send succeeded and can set the current sequence number equal to the one just received (in the next run we will expect a sequence number different from that one).

On the other hand, if the slave receives the same sequence number twice in a row, we must assume that our last packet was lost and that the master retransmitted the last sequence number to signal this condition. In that case we send our last answer again.

Note that the slave, unlike the master, does not have a time-out upon receiving data. This is perfectly logical: the slave should not be allowed to do anything at all if no permission (i.e., data) from the sender has been received, since this indicates an out-of-sync condition. A blocking of the slave until re-sync is the absolute right thing to do under such circumstances.

// set up
byte[] sendBuf = empty;
byte[] recvBuf = empty;
byte[] tempBuf = empty;
byte[] outBuf = empty;

int currSeq = 1; // slave starts with seq 1
int seq;

// slave loop
while (true) {

    // slave starts with receiving, NO timeout!
    tempBuf = receive data;
    seq = receive seq;

    if (seq != currSeq) { // last send was successful


if (tempBuf != ack && recvBuf != empty) {
    // provoke timeout, force master to resend
    goto begin;
}

currSeq = seq;

if (tempBuf != ack && recvBuf == empty) {
    recvBuf = tempBuf;
}

if (sendBuf != empty) {
    outBuf = sendBuf(currSeq);
    clear sendBuf; // release for new data
} else {
    outBuf = ack(currSeq);
}

// else: last send was not successful, resend
send outBuf;

Common packet handler methods

The following methods are offered by both the master and the slave. They are defined as absolute minimum by the API of a packet handler and allow applications/clients to make use of the duplex protocol. The sendingPossible method is not really necessary, as send will return false anyway if sending is not possible. But its usage may result in better understandable code.

Of course, synchronized (i.e., mutual exclusion) access to all of the signal variables must be used (i.e., to sendBuf and recvBuf).

boolean sendingPossible() {
    return sendBuf == empty;
}

boolean send(byte[] packet) {
    if (sendingPossible()) {
        copy packet into sendBuf;
        return true;
    } else {
        return false;
    }
}

boolean isDataAvailable() {
    return recvBuf != empty;
2.5 Communication between Brain and Spine

```java
int receive(byte[] buffer) {
    int size = 0;
    if (isDataAvailable()) {
        size = recvBuf.size;
        copy recvBuf into buffer;
        recvBuf.clear();
    }
    return size;
}
```

### 2.5.3 Addressing

We only have a single communication link which connects the main components of the Macrobot subsystems. Packets can be sent over this link in a full duplex fashion, thanks to the Janus protocol, which was introduced and explained in section 2.5.2. Although all communication is flowing through only two ports, from the iPAQ to the RCX and vice versa, the packets which pass over the comm link are not likely to be sent back and forth between the same two software components.

Rather, we expect that there will be packets originating from some components $B_A, B_B, B_C$ within the Brain, which will be sent as notifications of some sort to other components $S_A, S_B, S_C$ of the Spine. Examples of this might be request packets of the Rcx component which are being sent to the RcxServer, or SensitivityEvent messages originating at the MacroEngine and targeting the Executor.

Such message passing is a form of interprocess or interthread communication and requires addressing. Usually addressing is performed on the network layer of a protocol stack, because packets need to be routed from one machine to the other. In our case, packets must not be addressed with the target machine’s address, but rather with the target thread’s or the target component’s address. We thus put an extra addressing layer on top of the Janus protocol stack, which takes care of correctly addressing outgoing packets and which dispatches any incoming packets to the correct destination. We assume that a client component knows the exact address of the server component and that it hands this information to the packet handler below, along with the data to be sent to the desired destination.

### CommManagers

Within the Macrobot framework, so-called Communication Managers act as message or packet dispatchers between components. A single CommManager instance sits at each end of the comm link on top of the protocol stack. It properly formats the data it receives for sending to the other side into a specific packet format and labels the packet with the address of the receiver component. On the other side of the communication link, its peer receives the packet as soon
as it arrives and dispatches the payload data to the component that corresponds to the target address of the packet.

A component that wishes to send packets registers with the CommManager on its side and provides its own unique address along with a reference to itself. The CommManager saves the reference under the given address or rejects registration, if the given address already exists in its list. As soon as a packet arrives, the manager looks through its list of registered receivers and hands the packet to the appropriate component, if it finds a receiver with an address that matches the packet’s target address. If no such receiver exists, the packet is discarded.

Figure 2.3 illustrates the general concept of CommManagers. The graphic depicts two manager instances, each sitting on top of a Janus protocol stack. Client components are talking to the CommManagers only when wishing to send data across the IR link, just as discussed above.

CommManagers are not restricted to the Janus protocol, as a matter of fact they may sit on top of any communication stack, providing an inter-component communication service. All that a CommManager needs to function properly is a reference to the topmost packet handler of the underlying protocol stack. The following pseudocode sketches the functionality of a CommManager:

```java
// set up
PacketHandler lower; // topmost packet handler of protocol stack
Queue sendQueue = empty;
Queue recvQueue = empty;
Hashtable receivers = empty;

// CommManager loop
while (true) {
    // send pending packets
    while (sendQueue != empty) {
        remove first packet of sendQueue;
        lower.send(packet);
    }

    // receive packets
```
while (lower.hasPacketAvailable) {
    get packet from lower;
    enqueue packet in recvQueue;
}

// dispatch any received packets
while (recvQueue != empty) {
    remove first packet of recvQueue;
    if (packet.dest exists in receivers) {
        deliver packet to receivers.get(packet.dest);
    } else {
        discard packet;
    }
}

void send(byte[] data, int dest, int src) {
    create new packet;
    packet.data = data;
    packet.dest = dest;
    packet.src = src;
    append packet to sendQueue;
}

boolean register(Receiver receiver, int address) {
    if (address exists in receivers) {
        return false;
    } else {
        add (address,receiver) to receivers;
        return true;
    }
}

void unregister(int address) {
    if (address exists in receivers) {
        receiver = receivers.get(address);
        remove (address,receiver) from receivers;
    }
}

Of course, a concrete CommManager implementation can be extended with buffers or any other luxury that helps speeding up communication. This may be especially important if the communication through the given link proves to be a slow one. Buffering might prevent the blocking of client processes that wish to send packets. On the other hand, receivers should quickly process any arrival notifications, in order not to block the notifying CommManager thread.

As we will see in chapter 4, the shielding and abstraction of the communication link which is achieved by installing a CommManager on top of it, comes
Design

in very handy when implementing a simulator for the Spine.

2.6 DirectMode and MacroMode (vertical cut)

We recall that section 2.1 requires the Macrobot framework to provide means of executing macro programs. This requirement is fulfilled by the introduction of an own virtual machine, the Executor, capable of running macro programs within the Brain. It delegates the execution of the platform-dependent macros to the MacroEngine, sitting in the Spine.

However, section 2.1 also required that our framework must support rapid prototyping and testing of new robots, two requirements which can only be partially satisfied with the components just described. As a matter of fact, the requirements analysis literally demands that a low-level API must be provided, which allows direct access to the RCX’s hardware. This includes access to any attached sensors or motors, both reading and writing (if suitable).

The second request seems to be somewhat contradictory to the first one: on one side we try to achieve an abstraction from the robot’s hardware by means of macros, on the other side we still want to provide access to the guts of the machine. But again, we can explain the differing needs by restating our goal of building a general and open framework that supports the building, testing and deploying of new LEGO MINDSTORMS robots. By providing both high-level and low-level access to the robots that may become part of the framework, we ideally support every (or at least most) research on the topic. Such research may be performed on high-level subjects, such as testing and implementing the kind of adaptivity that chapter 1 talks about, or it may just be concerned with the development of a new robot architecture, in case of which low-level access is an indispensable feature.

We can compare the situation to that of testing a (new) car: at one moment we are using it to drive from location A to location B, steering it as we go along. At another moment we are inspecting the motor in the garage, having the car’s hood open and looking at its internals in order to understand how it works. We are working in two “modes” so to say, in “garage mode” and “street mode”, which are mutually exclusive, since nobody will attempt to drive a car with the hood open or working on a car’s mechanical devices while it is driving at 100 km/h.

Inspired by this analogy, the Macrobot framework also offers two modes for operation: On one hand we have MacroMode, which allows to execute macro programs and thereby implementing partially autonomous agents. On the other hand we have DirectMode, which, as the name suggests, allows direct access to the RCX’s internal and external components and which allows for control of a robot by issuing commands directly to its hardware.

Of course, those two modes are also mutual exclusive: switching from MacroMode to DirectMode immediately pauses the execution of any currently running macro program. Doing the opposite resumes operation of any halted macro program. Attempts to use features of either mode (such as loading a new program or calling a low-level function on the RCX) while the complementary mode is active will result in exceptions.

Both modes use a disjoint set of modules across Brain and Spine. The division into two modes thus effectively produces a vertical cut through the
2.6 DirectMode and MacroMode (vertical cut)

architecture of the Macrobot framework. The following two sections describe each of the two modes more closely and explain their design in detail.

2.6.1 DirectMode: Rcx and RcxServer

DirectMode is realized as an own client/server application within the Macrobot system. An abstract component, Rcx, defines the low-level API to be exported. A proxy component, called RcxProxy, forms the concrete implementation of Rcx. A singleton instance of this proxy is available from the Brain’s public interface and can even be invoked remotely over the network.

If an interface method is called, the proxy encodes a corresponding request packet which contains the ID and parameters of the invoked API method and sends this packet to the RcxServer module, which resides in the Spine. The server decodes the request and executes the desired function, using the LeJOS API. Any produced results (or none if the method is a void one) are piggybacked onto an ack-packet which is returned to RcxProxy. The original low-level message which was sent to the Rcx component then returns with the produced result. The exact format of the request packets is described in chapter 3.

If the server does not respond within a certain timeframe, the called method should signal a timeout error. In the case where a low-level function is executed while the Brain is in MacroMode, the proxy should dispose of the request and signal an appropriate error condition.

Table 2.3 lists all of the commands that are offered by the Rcx component and which form the low-level API of DirectMode. Many of those commands are very helpful when developing new programs for the RCX (freeMemory, for example, allows to get a good idea of memory usage while a program is running). Albeit their low-level nature it is very easy to build whole applications that, for example, use DirectMode to remotely steer a robot. Of course, such an application must normally be aware of the exact architecture of a robot, but often enough this is the case (especially in a research setting where only one type of robot is used). Appendix B presents some client applications that are based on DirectMode.

Adaptation in DirectMode

The abstraction which is established through Rcx and RcxProxy even makes adaptation of DirectMode applications a feasible target. It is very easy to change or replace parameter values which are passed to the low-level Rcx methods by inserting an extension at the beginning of the corresponding RcxProxy method. The change will be performed before the request is encoded, and an adaptor service thus needs no knowledge of the encoding format.

2.6.2 MacroMode: Executor and MacroEngine

We achieve a realization of MacroMode by defining two tightly coupled components, called Executor and MacroEngine. Both should be seen as virtual machines that execute abstract algorithms. But whereas the Executor, running as a separate thread within the Brain, interprets whole macro programs, it delegates the actual execution of the individual macro commands to the
MacroEngine which runs as part of the Spine. Due to the internal dual nature of the Macrobot architecture, resulting from being implemented across two platforms, the design of MacroMode is also inherently that of a client/server application. The Executor hereby acts as the client which sends macro requests to the MacroEngine.

The next subsection introduces the execution model of the Executor. This model defines the execution policy which is enforced by the Executor when running macro programs. After that, the actual design of the Executor and MacroEngine components are discussed in more detail.

### Execution Model

The Executor runs macro programs like the ones first mentioned in section 2.2.1 according to certain policies. Those policies are in turn specified by an execution model.

Programs that are to be run by the Executor are assigned a precedence, depending on their type. If a remote user loads a program with a higher precedence than the one which is currently running, the actual context is swapped out and execution of the new program takes place immediately. Once the new program has finished, the robot resumes the program which was previously swapped out.

The execution model which is discussed in this section defines the levels of

---

<table>
<thead>
<tr>
<th><strong>API method</strong></th>
<th><strong>Effect</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>batteryVoltage</td>
<td>Returns the current internal voltage of the RCX batteries in centivolt.</td>
</tr>
<tr>
<td>freeMemory</td>
<td>Returns the amount of free memory on the RCX in bytes.</td>
</tr>
<tr>
<td>activateSensors</td>
<td>Activates the selected sensors.</td>
</tr>
<tr>
<td>passivateSensors</td>
<td>Passivates the selected sensors.</td>
</tr>
<tr>
<td>readSensors, readSensor</td>
<td>Read the value of the selected sensor(s) with the selected mode, i.e. as boolean, percent or raw value.</td>
</tr>
<tr>
<td>sendMotorCmds, sendMotorCmd</td>
<td>Send one of forward, backward, reverse, float or stop to the specified motor(s).</td>
</tr>
<tr>
<td>getMotorStates, getMotorState</td>
<td>Returns one of forward, backward, float or stop for all or just the selected motor(s).</td>
</tr>
<tr>
<td>setMotorPowers, setMotorPower</td>
<td>Set the power of the selected motor(s) to a value between 0 and 7, inclusive.</td>
</tr>
<tr>
<td>getMotorPowers, getMotorPower</td>
<td>Return the power value of the selected motor(s).</td>
</tr>
<tr>
<td>lcdShowNumber</td>
<td>Displays a given value on the LCD display of the RCX, the position of the decimal point can be specified.</td>
</tr>
<tr>
<td>lcdShowProgramNumber</td>
<td>Displays a digit in the program section of the RCX’s LCD display.</td>
</tr>
</tbody>
</table>

Table 2.3: Low-level RCX API methods
precedence among those programs and describes the possible transitions among them. There are two precedence levels (basic and task) and three separate modes (normal, sensitive and handler) in which the interpreter may run. The Executor runs as a state machine and performs transitions between states that are combinations of precedence and mode values.

Let us first introduce the three types of (macro) programs that can be loaded into the interpreter. The description of those may already resolve some questions and give the reader a first idea of the Executor’s workings:

**Basic Behavior** There can only be a single basic behavior per robot, which is initially loaded and executed at startup time. A basic behavior program defines the actions that a robot carries out, if it has nothing better to do. Basic programs should never terminate, as this would leave the robot in an undefined panic state (i.e. not knowing what to do next). However, this doesn’t mean that a basic program must actually define any work to be performed. A program which does nothing else but waiting in an endless loop, leaving the robot in an idle state until a task (see below) is issued, is a perfectly reasonable basic behavior.

Basic programs may use any macro commands that are available and they can define sensitive blocks around any macro sequences within them, using sensitivity commands.

**Task** In contrast to basic behaviors, tasks are (normally) finite programs that eventually terminate. They have a higher precedence than basic programs and provoke pre-emption and swapping out of such if they are loaded into the engine.

Otherwise than the differences just stated, tasks are identical to basic programs and may also use any combination of macro commands and sensitivity commands.

**Handler** Handlers do not really have a precedence attached to them and they are exclusively executed in handler mode. Handler programs define the actions that a robot must take if a sensitivity interrupt occurs within a sensitivity block (i.e., in sensitive mode) during the execution of a basic program or a task. They can thus use any kind of macro commands plus an additional set of special handler commands, which define ways of how to continue the interrupted macro program.

The reason why handlers do not have a precedence is that they cannot be loaded externally, like a regular program, by a client. Rather they are set by macro programs (i.e., by tasks or basic behaviors) when opening a sensitive block. If an exception occurs at the sensor to which a handler is assigned, it forces the Executor into handler mode and thus pre-empts any currently running basic behavior or task. Any further sensitivity events (i.e., interrupts) are ignored in handler mode. Consequently, a handler program can not use sensitivity commands to define sensitivity blocks within itself. Not only because this would be meaning less in a state where all sensitivity interrupts are disposed of, but also because such practice would lead to (possibly) endless recursion.

The descriptions above mention four different command sets that can be used by the various program types. Table 2.4 explains their usage in general and
Normal Mode Normally the interpreter runs in this mode, i.e., it executes either a basic behavior or a task, without being sensitive.

Sensitive Mode This mode is activated as soon as the interpreter enters a sensitivity block. Sensitive blocks may be overlapping each other (see Figure 2.1 for an example), but entering a sensitive block when being in sensitive mode already does not change the current mode of the Executor. Sensitive mode terminates in favor of program mode, when a sensitive block is exited and no other sensitive block is still active.

Handler Mode The Executor enters into handler mode if it receives a sensitivity event from the MacroEngine while being in sensitive mode. Entering handler mode goes along with the execution of a predefined handler which was set by the interrupted program for dealing with the exceptional situation. Because handler mode signifies an exceptional state, no other programs, no matter how high their priority may be, will be executed and all interrupts (i.e., sensitivity events) will be ignored while it is active.

When the executed handler terminates, the interpreter will return to the state and mode it was in before the sensitivity interrupt was received.

The state machine which enforces the execution policy specified by the execution model is depicted in figure 2.4 below. It defines a start state for the Executor along with all the possible states and transitions between them. A state is described in the form programtype/mode. Table 2.7 lists and explains all of the transitions of the machine.

![Figure 2.4: A state machine that executes macro programs](image)
### Command Set

<table>
<thead>
<tr>
<th>Command Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro commands</td>
<td>Comprises of all the macros that are available within the framework. This set contains at least all of the commands listed in table 2.1.</td>
</tr>
<tr>
<td>Sensitivity commands</td>
<td>Contains two commands only. They are used for opening and closing sensitivity blocks.</td>
</tr>
<tr>
<td>Handler commands</td>
<td>Handler commands tell the Executor how to continue with the interrupted task or basic behavior. All of them force a termination of the handler and should thus only be used in places where a <code>return</code> statement would also be appropriate.</td>
</tr>
</tbody>
</table>

Table 2.4: Various command sets

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginSensitive(sensor, params, handler)</code></td>
<td>Opens a sensitivity block. The ID of the sensor to watch must be specified along with a parameter record and a reference to a handler (see section 2.3 for more information).</td>
</tr>
<tr>
<td><code>endSensitive(sensor)</code></td>
<td>Closes a sensitivity block. Sensor events generated by <code>sensor</code> will be ignored after this command.</td>
</tr>
</tbody>
</table>

Table 2.5: Sensitivity commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>redoMacro()</code></td>
<td>Terminates the handler and continues the interrupted program by re-executing the macro during the execution of which a sensitivity event was produced.</td>
</tr>
<tr>
<td><code>nextMacro()</code></td>
<td>Flushes the rest of the macro which caused the interrupt, then terminates the handler and continues.</td>
</tr>
<tr>
<td><code>continueMacro()</code></td>
<td>Finishes the handler procedure and resumes execution of the interrupted macro. In other words: the currently halted program is continued as if nothing had happened.</td>
</tr>
<tr>
<td><code>breakMacroSeq()</code></td>
<td>Exits the handler and continues the interrupted program after the sensitivity block which caused the interrupt. If the sensitivity block contained a single macro command only, then this command is equal to <code>nextMacro()</code>.</td>
</tr>
</tbody>
</table>

Table 2.6: Handler commands
Table 2.7: Transitions for the state machine shown in figure 2.4

<table>
<thead>
<tr>
<th>Transition</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic/Normal →</td>
<td>Entering a sensitivity block when in non-sensitive mode.</td>
</tr>
<tr>
<td>Basic/Sensitive</td>
<td></td>
</tr>
<tr>
<td>Basic/Normal →</td>
<td>Loading of task while running basic behavior.</td>
</tr>
<tr>
<td>Task/Normal</td>
<td></td>
</tr>
<tr>
<td>Basic/Normal →</td>
<td>Termination of basic behavior.</td>
</tr>
<tr>
<td>Panic</td>
<td></td>
</tr>
<tr>
<td>Basic/Sensitive</td>
<td>Leaving the last block of a possibly overlapping series of sensitivity blocks.</td>
</tr>
<tr>
<td>Basic/Normal</td>
<td></td>
</tr>
<tr>
<td>Basic/Sensitive</td>
<td>Loading of task while running basic behavior.</td>
</tr>
<tr>
<td>Task/Normal</td>
<td></td>
</tr>
<tr>
<td>Basic/Normal →</td>
<td>Sensitivity interrupt occurred while running in sensitive mode.</td>
</tr>
<tr>
<td>Basic/Handler</td>
<td></td>
</tr>
<tr>
<td>Task/Normal →</td>
<td>Entering a sensitivity block, i.e. processing a <code>beginSensitive</code> command.</td>
</tr>
<tr>
<td>Task/Sensitive</td>
<td></td>
</tr>
<tr>
<td>Task/Normal →</td>
<td>Termination of task, return to previous state.</td>
</tr>
<tr>
<td>Basic/Normal</td>
<td></td>
</tr>
<tr>
<td>Task/Normal →</td>
<td>Termination of task, return to previous state.</td>
</tr>
<tr>
<td>Basic/Sensitive</td>
<td></td>
</tr>
<tr>
<td>Task/Normal →</td>
<td>Leaving the last block of a possibly overlapping series of sensitivity blocks.</td>
</tr>
<tr>
<td>Task/Sensitive</td>
<td></td>
</tr>
<tr>
<td>Task/Handler →</td>
<td>Sensitivity interrupt occurred while running in sensitive mode.</td>
</tr>
<tr>
<td>Task/Sensitive</td>
<td></td>
</tr>
<tr>
<td>Task/Handler →</td>
<td>Handler has finished, return to previous state.</td>
</tr>
<tr>
<td>Task/Sensitive</td>
<td></td>
</tr>
<tr>
<td>Task/Handler →</td>
<td>Handler has finished, return to previous state.</td>
</tr>
<tr>
<td>Task/Sensitive</td>
<td></td>
</tr>
</tbody>
</table>

Executor

The previous section about the execution model of the interpreter already specified most of the Executor’s functionality by means of a state machine. However, it remains to specify the exact design of the Executor and its associated components along with the API that it exports. Figure 2.5 gives a rough idea of the modules that are involved.

The Executor runs each program within an own context that keeps track of the current mode and any other state information. A pre-empted program is swapped out onto the stack together with its whole context. A swap stack of size 2 is sufficient for all purposes, as we will never encounter a situation with more than two programs swapped out (with running handlers there are only three levels of precedence).

The Executor may be confronted with the situation where a client attempts to load a program of equal precedence to the one which is currently running. In case of tasks, this problem is resolved with a task queue in which incoming tasks are buffered if they can not be run right away (e.g. if another task is running or if the Executor is in handler mode). If the user attempts to load a
2.6 DirectMode and MacroMode (vertical cut)

basic program, the currently running basic program will be replaced with the new one. At startup, the Executor starts with an idle basic behavior, i.e. does nothing. Generally, when a program has run to its end, the interpreter checks queues for any more programs that bear the same precedence. New programs will be loaded one by one until no more are available.

Clients should use the loadBasic and loadTask methods of the Executor to load a new program into the virtual machine. Freezing (i.e., pausing) and resuming of the interpreter’s operation is achieved by sending it a halt or go message, respectively.

The Executor can run in MacroMode only. If the Macrobot framework switches to DirectMode by request of a client, the operation of the interpreter will be frozen at once and will not be resumed until returning to MacroMode. Invoking any of the methods exported by the Executor interface while DirectMode is active will result in an error.

MacroEngine

Within the Spine, the MacroEngine component is the virtual machine that fetches, loads and executes macros. It does so upon requests that are sent to it by the Executor. The Executor is both a controller and a client of the MacroEngine. Such a request usually contains the unique ID of the macro to execute along with any necessary parameters.

The engine fetches the executable macro unit corresponding to the provided ID and starts it. Once the macro is running, the following may happen:

1. The macro terminates without being interrupted, in which case the macro engine sends an ack message to the executor. This is the normal case.

2. The macro program which invoked the currently running macro happens to be pre-empted in favor of a higher-precedence program. In that case, the Executor will immediately send a "pre-empted" message to the Macro-Engine. The latter should then halt the running macro at once and save its state. After that it may execute any new macro requests. As soon as the program that caused pre-emption terminates (which will be signalled), the engine will reload the previously stopped macro and continue to execute it as if nothing had happened.
3. The SensorWatch, which is an associated component of the MacroEngine, might signal a sensitivity event. Should this happen, then the engine will notify the Executor at once of this condition, halt all operation and save the state of the current macro. Sending a sensitivity event to the program interpreter will cause it to swap out the current program, to go into handler mode, and to execute the appropriate handler. Once the handler has finished (possibly by issuing any of the special commands that are listed in table 2.6), the MacroEngine resumes the operation of the saved macro (unless it was flushed by the handler).

The MacroEngine also translates the `beginSensitive` and `endSensitive` commands into action by registering or unregistering for the appropriate events at the SensorWatch.

In general, the MacroEngine always halts its operation when it is notified about a context change within the Executor, performing a synchronous context change on the macro level. That is, the engine saves the state of any not finished macro for completion at a later time.

### 2.7 Complete System Overview

![Diagram](image)

Figure 2.6: Overview of the Macrobot framework.

Figure 2.6 gives a schematic overview of the Macrobot system. One can clearly identify both the vertical cut (separating DirectMode and MacroMode).
as well as the horizontal cut (separating Brain and Spine), the latter being connected by the framework’s proprietary Janus protocol stack.

The picture indicates the main communication links within the system. Messaging across the IR-link appears transparent for the components involved. We can see that both Executor and RcxProxy as well as MacroEngine and RcxServer are governed by the Brain- and SpineController, respectively.

Abstraction from both the client space and the robot hardware is realized by the BrainController API on one side and by means of the LeJOS classes on the other.

There’s also a Ping object within the SpineController. This component stands vicariously for the extra services that the Spine offers to the Brain, e.g. for configuration or testing purposes.

2.8 Hardware Architecture (Lego-Robots)

2.8.1 General Requirements

The physical construction of any robot deployed within the Macrobot framework must adhere to some minimal requirements.

First of all such a construction must integrate the iPAQ, preferably in a way so that the pocket computer’s display is accessible (allowing, for example, to program a visual interface to the Brain) and so that the handheld can easily be inserted into and removed from the robots body. This requirement has severe consequences for the design of a robot, as the iPAQ, together with its sleeve (containing a microdrive, a wireless LAN card and an extra battery pack), has dimensions of about 8.5 x 13.5 x 4.5 cm and weighs a little more than 400 grams, making almost every construction appear bulky and less manageable, both in terms of size and maneuverability.

Secondly, the RCX must be integrated into the robot, of course. Again there are constraints that affect the design: the programmable brick should also be placed so that its console can be accessed, especially the on/off button should be reachable comfortably. Also the RCX must be placed in such a way that it either faces the IR-tower’s front side (the tower being a necessary component for integration into the robot’s construction as well) or so that it faces the IR port of the iPAQ. The latter solution is feasible only, if an implementation of Macrobot is available which takes advance of the rcxir package, a library that eliminates the tower by using the built-in IR ports of a laptop or iPAQ for communication with the RCX.

With all of the mandatory components just listed, an actually built robot inevitably gains a certain weight. A consequence of this is, that the amount of inertia that a robot displays becomes larger as well. In other words, maneuverability is affected and the robot won’t be able to stop or accelerate as fast as it would if it were lighter.

The power that a motor produces is is usually transformed with a bias for either torque or speed. The increased weight of a Macrobot will lead to architectures that sacrifice speed for torque, in order to move the robot. Also, because of the extra amount of inertia, robustness of constructions will have to be increased. A heavy robot crashing into a wall tends to break more easily than a light one.
Last but not least, the *Macrobot* framework is designed for robots that are equipped with at least one sensor. Therefore, all robots should best integrate two or three sensors. Currently all sensors are read in percent mode, so it doesn’t really matter what types of sensors one uses. However, sensitivity can be set with respect to single edge detections only, it is not possible to register for events produced after a minimum of edge counts. This makes it somewhat unreasonable to use rotation sensors or similar devices.

### 2.8.2 Lego Mechanics

Constructing robots with *LEGO MINDSTORMS* is fairly simple, even for the inexperienced. However, a minimum knowledge about LEGO mechanics (brick size ratio, joints, available pieces, building strategies) and mechanics in general will greatly improve both the quality and functionality of new constructions.

For the "mechanically inclined" we recommend [2] as an excellent introduction to the world of *LEGO TECHNICS* with a focus on *LEGO MINDSTORMS*. [3] is a shorter but also a very good source of information.
Chapter 3

Implementation

This chapter discusses the details of the implementation of the Macrobot framework. To some extent, it follows chapter 2 both in structure and content. However, only the implementation of the core framework is discussed here. The realization of the Macrobot simulator is commented separately in chapter 4. See appendix B for information about various client applications.

3.1 Macros and Macro Programs

Macros and macro programs are a central feature of the Macrobot framework. By design, the system must offer a way to execute programs or scripts composed from individual macro commands. Chapter 2 conceptually draws the picture of two virtual machines that accomplish this task, one within the Brain for running macro programs and the other as part of the Spine for the execution of single macros in hardware.

3.1.1 An Interpreter for Macro Programs

For an implementation of a virtual machine that executes macro programs within the Brain, we can think of three solutions that are favorable. Each of them has its own advantages and disadvantages:

1. We can actually design an own scripting language, write a parser for it and implement an interpreter of some sort. The interpreter will execute some intermediate, machine-readable representation of the commands that we have defined within our language.

This approach has several major advantages: We are completely free to define both the syntax and the semantics of the language. By writing our own interpreter we can truly control every step of the execution, which makes it very easy to realize pre-emption and context swapping, for example. The definition of a client interface for loading new programs becomes very thin and clean. In its simplest form, it only exports a single method taking but a string (the program code) as its only argument. This leads to a very high degree of encapsulation: the only information we need to make available to clients is the grammar of the scripting language. With
42 Implementation

such a scenario, networking capabilities of the Brain become very easy to implement, as we can employ any protocol that supports the transmission of simple ASCII files.

There are also drawbacks of the approach: the load we put on the Brain increases, since we must add the whole compilation unit (consisting of the usual modules such as Lexer, Parser and Compiler). We could supply the client with those tools, but then the simplicity of our interface will decrease and the extent of encapsulation will be reduced. Furthermore, the development of an own scripting language is usually an error-prone process, even with automatic parser and compiler generator tools at hand.

2. We can model macros and macro programs within the Java language and use the JVM for interpretation.

Again, a lot of advantages immediately come to our mind: the writer of macro programs can use a language she is familiar with. This is an advantage because (i) there is no new syntax to learn and (ii) because we get to use the expressive power and the extensive libraries of a well-established, object oriented high level language at no extra cost. Also, the amount of extra work on the compilation unit is reduced to zero. A careful design will, due to the object oriented nature of the language, allow us to add further commands and features without too much trouble.

The disadvantages of this solution are obvious: controlling the execution of the programs provided by a client is much harder, because they run as own threads on the Java virtual machine, about which we have only limited control. SUN has deprecated all of the thread controlling methods (for perfectly rational reasons), which leaves us with a non-trivial problem when trying to stop and restart a program thread (e.g. in a pre-empt situation where we must temporarily halt a a program for the time it will be swapped out). Another inconvenience is that we must export many more components in order to give clients the opportunity to implement their own programs. Encapsulation will be suffering to some extent. Still we certainly do not wish to expose the internals of the whole framework; hence the interface between the interpreter (or rather, executor) within the framework and the client programs that are loaded must be well abstracted and allow a clean detachment of the two components. As this solution treats macro programs as objects, we would have to think more carefully about how to load programs over the network. Yet with Java being a very networking-friendly language, this issue is more of theoretical interest and does not really pose any problems in reality.

3. We could use the parser and compiler front-end of an existing scripting language, of which an implementation under Java is available (e.g., jython), and implement our own back-end that puts the compiled programs into action.

This solution is a blend of the first two but after a closer analysis, it mainly seems to combine their disadvantages without providing any reasonably better advantages. Hence we did not further explore it.

With solution 3 being rejected, we tend to chose the first one of the remaining two alternatives, because it offers better control of execution. However, with
the time constraints of this thesis in mind, solution 2 seems more workable and thus wins the race.

But before we can go on discussing the exact realization of the macro program interpreter we should first devise a solution for the related task of the MacroEngine component, which is to run Macros within the Spine.

### 3.1.2 Executing Macros

On a lower level, the MacroEngine performs the same chore as the Executor, only that the unit of execution is a single macro rather than a whole program. The engine is responsible for translating a single abstract macro command into an architecture-dependent series of control commands that steer the hardware of a robot. Therefore we need yet another virtual machine that runs macros.

However, due to the heavy memory constraints of the RCX that are imposed on every component realized within the Spine, the set of alternatives from above is not really applicable here. Writing an own interpreter would consume too much of our resources and also introduce extra overhead we can not really justify. Here, we are forced to find a solution that can be realized with Java (or rather LeJOS), and thus end up choosing a similar strategy as we did for the implementation of the Executor.

### 3.1.3 Modeling Macro Programs in Java

We present two solutions for modeling macro programs in Java:

1. Using the command or interpreter pattern in combination with the composite pattern, the user can assemble programs from single command objects. Such a program is not really an executable itself, but rather an object oriented model of the algorithm to execute. Its components (i.e., commands) have to be interpreted one by one.

2. The actual algorithm is specified by overriding or implementing some specific method of an abstract program class. Macro commands are issued by calls to the API exported by the same class. The extended program component implements the Runnable interface and can later be run within an own thread.

Solution 1 is abandoned, because it makes it just too complicated and tiresome for the programmer to write new programs for obvious reasons. Of course the programming process could be supported with extra tools, but eventually we would end up writing a parser that composes the program structure for us, which is what we already decided not to do.

Solution 2 is more elegant: the programmer can use normal Java syntax to create new programs. Implementing an algorithm as an own method is a logical thing to do and allows for creation of arbitrarily complex programs, using any of the JDK’s API classes, if necessary. By creating an inheritance hierarchy like the one depicted in figure 3.1 we assign different types to different programs and can make sure that no program is using any special commands that it shouldn’t.

The following code example shows how a basic behavior can be implemented using the program-model illustrated by figure 3.1:
public class Explorer extends Basic {

    int FRONT_BUMPER = S1;
    Handler bouncer = new BounceOff();
    SensitivityParams bumperParams = new BinaryEvent();

    // this method should never terminate (basic behavior)
    public void programCode() {
        // be sensitive with respect to front bumper
        beginSensitive(FRONT_BUMPER, bumperParams, bouncer);

        // keep going forward forever
        while (true) {
            forward(Integer.MAX_VALUE);
        }

        // not really necessary (will never be reached)
        endSensitive(FRONT_BUMPER);
    }

    // inner class that implements a handler
    public class BounceOff extends Handler {

        Random rand = new java.util.Random();

        public void programCode() {
            // set back a bit
            backward(10);

            // make a turn between 90 and 270 degrees
            turnLeft(rand.nextInt(180)+90);

            // this is default, but we still specify
            continueMacro();
        }
    }
}

The program assumes a front-bumper architecture, with sensor S1 being the
one attached to the bumper. It will drive the robot around a room until it runs
into a wall or another obstacle. Upon such an event, the observed sensor sends
an interrupt and the associated handler (assigned with the beginSensitive
command) will be executed. As a result, the robot will back up 10 cm, make a
turn away from the wall and then continue with its exploration.

The example illustrates very well how simple the implementation of a new
program can be. A program can use any Java features it wishes to take ad-
vantage of. A few examples are JDK API calls (like the call to Random in the
example), exception handling, or the wealth of Java's data structures. Regular
Java commands can be tightly interweaved with macro commands, e.g., to
maintain an inner state of the robot. It may be good practice to implement
3.1 Macros and Macro Programs

For the implementation of macros we did not choose to run them as separate threads, instead they will be run within the MacroEngine’s own thread once they’re about to be invoked. Macros are implemented by extending the abstract Macro class which is shown in figure 3.2. All macros are kept as singleton instances and are managed by a MacroFactory component within the spine. Calling a macro command in a program causes the Executor to send the ID of the corresponding macro to the MacroEngine, which in turn fetches the singleton macro instance at the MacroFactory.

The macro itself is then executed in slices, i.e., tiny, identical parts of the macro that perform only a small percentage of the complete command. Between each slice, the macro checks if it has been interrupted (due to a sensitivity event), and if so, it returns control to the main thread of the MacroEngine, along with the percentage that it completed. If that percentage is less than a 100%, the macro will be re-executed with (100%-completed) if a continue command should be issued by a handler thereafter. The redo behavior is just
Implementation

as simple to implement: here, we just call the macro’s `execute` method with 100% again, indicating that we wish to execute the macro until 100 percent have completed.

A macro has set-up and tear-down methods called `init` and `finish`. They are called every time a macro is invoked and before control is returned to the MacroEngine, respectively, but never between slices. A concrete macro implementation overrides those methods to initialize and start motors, for example, or to stop them.

A less sophisticated example of a macro implementation is given in the following code. It implements a forward movement about a given distance and is based on timing. Constant speed is assumed and acceleration is not taken into account.

```java
public class ForwardMacro extends Macro {

    final int SLICE = 5; // sleep interval in ms
    final double VELOCITY = 0.0412; // 41.2 cm/sec
    long TIME_0, TIME_1;
    int DISTANCE; // to be initialized

    public float macroSlice() {
        // avoid division by 0
        if (DISTANCE == 0) return 100;

        try { Thread.currentThread().sleep(SLICE); }
            catch (InterruptedException e) {}}
        TIME_1 = System.currentTimeMillis();

        // ts/t * 100 = perc per slice
        double perc = ((TIME_1-TIME_0)*100/(DISTANCE/VELOCITY));
        TIME_0 = TIME_1;

        return perc;
    }

    public void init(int distance) {
        DISTANCE = distance;
        Motor.A.setPower(6);
        Motor.C.setPower(6);
        Motor.A.forward();
        Motor.C.forward();
        TIME_0 = System.currentTimeMillis();
    }

    public void finish() {
        Motor.A.stop();
        Motor.C.stop();
    }
}
```

The `execute` method, which is part of the abstract Macro class, calls `macroSlice`
3.2 Sensitivity

until the cumulative return values of this method add up to more than a 100. The parameters for the macro are handed to the init method and are used to set up the macro. We can see that the program makes use of the Motor class (which is part of the LeJOS API) to control the motors of the robot.

Because macro implementations are architecture dependent, they usually contain hard-coded facts about the robot they are programmed for. The code above, for example, is based on the (measured) velocity of a specific robot, if its motors run at power 6. Obviously the target architecture covers about 41 cm per second when driving forward; this fact has been encoded in the global VELOCITY constant.

Appendix C gives more hints and details about the implementation of macros within the Macrobot framework.

3.2 Sensitivity

Sensitivity is realized with a singleton SensorWatch object, a poller thread which constantly reads the values of all sensors that are attached to the RCX. Event notification employs the observer pattern.

The MacroEngine registers for the events of a specific sensor at the SensorWatch, if it receives a beginSensitive command, and unregisters again, when endSensitive is encountered in program code. Registering is performed according to a set of parameters that specify a filter for sensitivity events. SensorWatch keeps each registration in a list of registered ChangeListeners, along with the supplied ListenerParameters. If a newly read value passes one of the provided filters, the associated listener (which is always the MacroEngine) is notified with a valueChanged message. There can be a maximum of three registrations, one for each sensor. Once a registration is removed, the corresponding listener will no longer be notified of any events that are generated by the sensor that it had previously registered at.

Figure 3.3 illustrates the sensitivity implementation within the spine package.

If the MacroEngine receives a sensitivity interrupt from the SensorWatch, it immediately halts the currently running macro (if there is one). Then the
48 Implementation

Engine encodes a sensitivity event packet and sends it to the Executor. Finally it changes into handler mode, awaiting new commands from the Brain that tell it what to do next. The format and encoding of the sensitivity event packet is depicted in figure 3.4.

![Figure 3.4: Packet format for sensitivity events](image)

Because sensitivity is inactive when the macro program interpreter goes into handler mode, the MacroEngine also stops the SensorWatch for the duration of handler mode. As soon as the handler is done, the watch is restarted and the state before the exception is restored.

3.3 Brain and Spine

Brain and Spine have been implemented as two separate packages. The components of those packages communicate with each other by using the services of an extra communication package. The functionality and contents of the latter are described in section 3.4.

BrainController and SpineController form the main entry points of the Brain and the Spine application, respectively. Both controllers implement a more general Controller interface.

The classes of the ch.ethz.inf.macrobot.brain package (and sub-packages thereof) are intended to be run on a regular JVM, whereas the components of the ch.ethz.inf.macrobot.spine package will exclusively run under LeJOS.

3.4 Communication

All classes that are utilized for framework-internal communication are logically bundled in the ch.ethz.inf.macrobot.comm package. This package is shared by the Brain and the Spine applications.

3.4.1 Hardware Dependent Classes

Because Macrobot is a framework which is spread across two different hardware platforms, we may encounter the problem of having to implement two different versions of the same class. Within the comm-package this is the case for the JanusHandler class which implements the Janus protocol layer just above the hardware-dependent link layer. This class makes use of two different link layer classes, depending on the platform it runs on, i.e., of josx.rcxcomm.LLC on the RCX and josx.rcxcomm.Tower on the PC/iPAQ.

LeJOS knows the same problem with respect to its own protocol stacks, which also employ those two classes as their lower ends. Here, the developers solved the problem by creating two different releases (i.e., two different jar-files) of the josx.rcxcomm package. Applications which run on the PC include
3.4 Communication

pcrcxcomm.jar on their classpath, whereas LeJOS by default uses the classes that are contained rcxrcxcomm.jar.

For the deployment of the ch.ethz.inf.macrobot.comm package we chose an identical approach. The files of this package are stored in three different physical locations, depending on whether they are truly shared (and thus platform independent), or exclusively used on the Spine or the Brain platform. For the actual release, we pack all of the shared classes together with the RCX classes into one jar-file and all of the shared classes together with the PC classes into another one. The Spine and Brain applications then include the custom-made jar-file on their respective classpaths.

As a matter of fact, this three-faced deployment policy is used throughout the whole Macrobot project. Any package that contains platform specific classes is partially replicated in three different locations, according to the scheme just described.

3.4.2 Janus Protocol Stack

The Janus protocol stack is composed of 5 logical layers which are illustrated by table 3.1 along with their main function. Each layer, with the exception of the link and the addressing layer, is implemented by extending the josx.rcxcomm.PacketHandler class. Let us quickly discuss the implementations of the various layers:

<table>
<thead>
<tr>
<th>Layer name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing</td>
<td>Receives and dispatches addressed data</td>
</tr>
<tr>
<td>Duplex</td>
<td>Synchronizes communication</td>
</tr>
<tr>
<td>Integrity</td>
<td>Ensures integrity of the payload data</td>
</tr>
<tr>
<td>Packet</td>
<td>Enforces the outer Janus packet format</td>
</tr>
<tr>
<td>Link</td>
<td>Sends and receives raw bytes</td>
</tr>
</tbody>
</table>

Table 3.1: Layering and functions of the Janus protocol stack

The link layer is provided by LeJOS and consists of either the Tower or the LLC class. Packet and integrity layer have been merged into a single PacketHandler implementation for economical reasons, but are otherwise heavily based on the LNPHandler and LNPIntegrityHandler implementations of the LeJOS josx.rcxcomm package. The duplex layer runs in an own thread and implements the master/slave protocol which was devised in section 2.5.2. The actual implementation consists of two different classes, a slave class and a master class. The addressing layer consists of a CommManager implementation and runs also in an own thread.

Each layer adds control information to the data which is supplied at the top of the stack, continuously enlarging the byte frame which is finally sent off at the link layer. The function and position of the extra bytes of each layer is illustrated in figure 3.5. In total, five bytes of extra information are added by the Janus stack layers: header, length, seq-number, address and checksum. Because length is a byte, the payload size of the outermost frame is restricted to a maximum of 255 bytes. Depending on how addresses are encoded, this leaves a maximum size of 253 bytes for a data packet handed to a comm manager.
Implementation

(assuming that the address range is restricted to 0..15 and that both source and destination address are packed into a single byte, as shown). This maximum size is by far more than we’ll ever need.

<table>
<thead>
<tr>
<th>src/dest address</th>
<th>data bytes</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence number</td>
<td></td>
<td>Addressing</td>
</tr>
<tr>
<td>length</td>
<td></td>
<td>Duplex</td>
</tr>
<tr>
<td>prefix</td>
<td>checksum</td>
<td>Integrity/Packet</td>
</tr>
</tbody>
</table>

Figure 3.5: Extra frame information added by protocol layers

Analysis of the inherent peculiarities of the IR link between the Tower and the RCX and of existing protocol layer implementations (LNP protocol stack) turned up two problematic issues that had to be resolved. First of all, there seems to be a problem with the serial tower as it reads any bytes that it sends right in again\(^1\). The packet layer should take care of this problem. It does so by prefixing every packet with a specific start byte, which is different from the peer station’s start byte. Any packets that do not start with the expected bit sequence are discarded on this level. Secondly, the existing protocol implementations use rather generous buffers on each level of the protocol stack. For economical reasons, concerning memory on the RCX, we are implementing our own protocol stack in such a way that all layers can be configured for a maximum packet size. Also, packet objects are only instantiated once and then reused throughout the lifetime of the application.

A last word about the parameterization of the duplex master class implementation: It is crucial to select a proper timeout value for this component. Although the protocol specifies where the timeout should be adjusted if it is too short, it is not an easy task to choose a sensible initial value. If it is too large, the communication will be slowed down in case of frequent collisions. If it is too small, it will result in further failures. Best practice might be to determine the timeout value by detecting the roundtrip time of some packet of maximum length empirically and then add some extra safety time to the roundtrip time. Currently the value is set to 200 milliseconds, which works fine.

3.4.3 CommManagers

The concept of CommManagers has been discussed in section 2.5.3. For the implementation, the general services of a CommManager are defined by an abstract class, which has been subclassed multiple times. There are implementations that support unlimited buffering and registrations, others that are very limited in their resources and even others that were mainly built to support testing and simulation. Only one of those implementations can be used on the Spine but all of them can be employed by Brain applications. Instantiation is

\(^1\)Reportedly, the USB tower does not show this behavior, it seems to be a peculiarity of the serial tower. However, the LeJOS support for the LEGO USB tower under Linux is still not satisfactory at the time of writing. Hence it was not feasible for us to use an USB tower for the prototype implementation.
normally performed through a factory which automatically supplies the CommManager(s) of the desired type with a correctly instantiated comm stack.

Since a CommManager is both a message source and a message sink (i.e., packets are both dispatched and collected) implementations are based on the observer pattern. Any process or component that wishes to send or receive messages has to come up with an unique address (that is, unique with respect to the address space of the CommManager) and register at the manager. Registration succeeds if the address is not in use by any other currently registered component. It is perfectly legal for a component to register itself two or more times under a different address. This may be very convenient in some cases, because it gives a single object the chance of collecting the messages for logically distinct destinations. Receivers must implement the DataReceiver interface. Upon notification through the packetArrived method they may call back to the CommManager to claim their data, providing a reference to an empty packet of sufficient size, which is then initialized with the data. Figure 3.6 depicts the implementation of the CommManager and related components.

Within the Macrobot framework, CommManager implementations never keep any reference to packets they are passed for sending or receiving, because components are usually reused to save memory. Instead, the contents of the sender’s packet are copied into own buffers and vice versa. Thus, when the send method of a CommManager returns, the sender is free to reuse the packet without having to worry about corrupted data.

### 3.5 DirectMode

Direct mode is realized as an own client/server application within the Macrobot system. An abstract class, Rcx, offers the complete low-level API of the RCX programmable brick. Here, the most important functions of the API of the josx.platform.rcx package are packed into one single unit. This unit is implemented by RcxProxy, which translates all of the API calls into request packets that are sent to the RcxServer component running on the RCX. Throughout the whole Macrobot framework there is just a singleton instance of the Rcx class available, which can be accessed through the getRcx method of the BrainCon-
troller. It can be considered as a substitute for the actual physical RCX brick, which forms the Spine of any particular MINDSTORMS robot deployed within the Macrobot system.

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>batteryVoltage, freeMemory, getMotorStates, getMotorPowers</td>
</tr>
<tr>
<td>2</td>
<td>activateSensors, passivateSensors</td>
</tr>
<tr>
<td>3</td>
<td>readSensors, setMotorCmds</td>
</tr>
<tr>
<td>4</td>
<td>setMotorPowers</td>
</tr>
<tr>
<td>5</td>
<td>lcdShowNumber, lcdShowProgramNumber</td>
</tr>
</tbody>
</table>

Table 3.2: Command groups for DirectMode requests

The API of the Rcx class was introduced and defined in section 2.6.1. For the implementation, the DirectMode API commands have first been subdivided into the command groups that are listed in table 3.2. Then, a packet format has been devised for each group, which encodes the parameters for the commands of that group in the least amount of bytes possible (for one reason, memory is a precious resource and for another, we want to keep packet size down so as to speed up communication). Figure 3.7 shows the packet formats for each command group.

All packets bear the ID of the command that they encode. This information is always contained in the lower nibble of the first packet byte. For multi-request commands (e.g., commands that get results from multiple sources) a 3-bit-mask selects the sources to which the command will actually be applied.
3.5 DirectMode

The RcxServer executes the requests received from the RcxProxy and returns the results of the API command (if it produces any) by piggybacking them onto an ack packet. The various ack-packet formats are depicted in figure 3.8, table 3.3 shows which format belongs to which command.

Requests are always acknowledged by the server, even if they do not produce any results. The absence of an ack within a certain amount of time can thus provoke a runtime TimeOutException in calls to the Rcx component. The maximum timeout period has been set rather generously, so that hardly ever any timeout exceptions are thrown, except for the case where the RCX is not reachable for some reason.

<table>
<thead>
<tr>
<th>Format ID</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>batteryVoltage</td>
</tr>
<tr>
<td>2</td>
<td>freeMemory</td>
</tr>
<tr>
<td>3</td>
<td>readSensors</td>
</tr>
<tr>
<td>4</td>
<td>getMotorStates, getMotorPowers</td>
</tr>
<tr>
<td>5</td>
<td>activateSensors, passivateSensors, readSensors, setMotorCmds, setMotorPowers, lcdShowNumber, lcdShowProgramNumber</td>
</tr>
</tbody>
</table>

Table 3.3: DirectMode commands with identical ack format

Yet another runtime exception, WrongModeException, is thrown by the Rcx methods, if an user tries to use DirectMode services while MacroMode is active. As we know from section 2.6, those two modes are required to be mutually exclusive by design.
3.5.1 DirectMode Example

The following piece of code is taken from a client application which uses DirectMode to steer a robot. The code was written for an architecture with a differential drive, i.e., the machine has two separate motors powering a single wheel each. The robot drives forward by running both motors with equal power into the same direction; turns are performed by driving one wheel forward and the other one in reverse direction. We further assume that there is a bumper attached to sensor S1. If the bumper is pressed, its boolean value changes from false to true.

```java
// make the robot move forward
Rcx rcx = brain.getRcx();
rcx.setMotorPower(rcx.POWER_3, rcx.IGNORE, rcx.POWER_3);
rcx.sendMotorCmds(rcx.FORWARD, rcx.IGNORE, rcx.FORWARD);
rcx.activateSensor(rcx.S1); // not really necessary

// poll S1 until a change is detected
int sensorValue = 0;
while (sensorValue < 1) {
    sensorValue = rcx.readSensor(rcx.S1, rcx.BOOLEAN);
}

// make the robot spin
rcx.sendMotorCmd(rcx.A, rcx.REVERSE);
```

Any architecturally suitable robot would first drive forward until the bumper is pushed, then spin in place counterclockwise.

Actually, it is not necessary to activate touch sensors, but it doesn’t do any harm either. We included the statement for the sake of the examples generality. Figure 3.9 illustrates the message passing between the DirectMode components when executing the given piece of code.

Please consult the API documentation of the Rcx class for a more detailed tutorial on how to use its services.

3.5.2 Controlling DirectMode

DirectMode is selected by default when the Brain application is started. However, during the lifetime of the application a client can switch between DirectMode and MacroMode at any time by sending either a selectDirectMode or a selectMacroMode message to the Brain, respectively. If the user changes to MacroMode while a direct mode request is still pending, then the pending call will return with a WrongModeException and any ack that arrives at a later time will be discarded. The BrainController takes care of synchronizing the Spine, because it receives the selectMacroMode message in the first place.

3.6 MacroMode

The program- and execution models of MacroMode were already discussed in sections 3.1.3 and 2.6.2. This section is thus more concerned with the workings
3.6 MacroMode

Figure 3.9: Processing of a DirectMode request

of the **Executor** and the **MacroEngine**, which are responsible for the execution of macro programs and macro commands, respectively.

Similar to DirectMode, MacroMode has been implemented as a client/server architecture, with the Executor being in the role of the client requesting the execution of single macro commands by the MacroEngine which acts as the server. Although, in a classic scenario, the server does never take initiative by itself, this is not always true for our setting: in case of a *sensitivity event*, the server informs its client spontaneously of that condition, demanding a change to *handler mode* along with the execution of an appropriate handler procedure. Thereafter it reverts to its serving function.

### 3.6.1 Executor

The Executor is very loosely coupled with the macro programs that it executes. The reason for this is that we absolutely want to minimize the number of classes needed for creating and compiling macro programs. By following this strategy, we also keep the encapsulation of the framework’s internals as tight as possible. The programmer of macro algorithms bases all of her concrete program implementations on abstract classes and interfaces. Upon loading of such a compiled
program, the Executor replaces any references to abstract components with concrete implementations. Figure 3.10 depicts the general implementation design of the Executor and its associated components. The parts of the graphic which correspond to the program model (which is depicted in in figure 3.1) are only minimally rendered.

![Figure 3.10: Implementation of the Executor](image)

Different macro program types implement different interfaces, depending on the commands that they export. But the implementations of those commands are actually provided by other classes, namely the CommandsProxy classes within the Brain. The different abstract macro program implementations act only as *facades* for those latter classes and forward all calls to them. When a macro program is loaded, the Executor sets the references to the proxies before it runs the macro program. The proxies encode all calls to the various commands into packets which are then sent to the MacroEngine over the comm link. Most of those commands contain only the ID of the macro to execute, along with a single integer parameter (or alternatively two short parameters).

Table 3.4 and figure 3.11 show the encoding of the commands that are sent to the MacroEngine by the Executor.

<table>
<thead>
<tr>
<th>Group ID</th>
<th>Encoded Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>administrative commands, such as <code>begin</code> or <code>done</code>, as well as all handler commands (do not bear any arguments)</td>
</tr>
<tr>
<td>2</td>
<td><code>sensitivityOff</code></td>
</tr>
<tr>
<td>3</td>
<td><code>sensitivityOn</code></td>
</tr>
<tr>
<td>4</td>
<td>macro commands such as <code>forward</code>, <code>backward</code>, <code>turnLeft</code> or <code>turnRight</code></td>
</tr>
</tbody>
</table>

Table 3.4: Command groups for *MacroMode* requests
3.6 MacroMode

All of the requests that are sent by the Executor are acked by the MacroEngine as soon as they are processed. The format of the ack-packets is depicted in figure 3.12. It is very simple and basically just returns the ID of the command that has been processed. The run level (i.e., state of the MacroEngine) at the execution point of the command is also reported, for control purposes only. Solely for the execute command (i.e., request for the execution of a macro) the ack additionally bears a flag that indicates whether the macro has completed or whether it has been flushed (aborted) before completion. The latter may have been caused by handler commands such as next or breakMacroSeq.

Apart from the ack-packets, there is another message that may be sent to the Executor by the MacroEngine: a SensitivityException. The format of this packet has already been discussed in section 3.2 and is depicted in figure 3.4.

Once a program has been loaded, it is run within its own thread. If regular Java code is executed, the Executor cannot take any influence or control. However, when a macro program specific command is executed (i.e., a command specified by one of the Command interfaces), then various signals are checked before the command is actually executed. Thus we are able to pause, resume or kill macro programs.

If a pause signal is encountered by a command proxy, it goes into waiting until it is signalled to continue. If a kill signal is sent, the detecting proxy throws an exception which will not be caught until within the execute method of the abstract MacroProgram class where it will lead to a ”regular” termination of the program thread.
Each program type is associated an own set of command proxies with different addresses, thus the communication with the MacroEngine is layered with respect to the current execution level/mode of the Executor.

If a program with a higher priority is loaded to the Executor, the interpreter first sends a pause signal to the currently running macro program (it does not matter whether this signal is detected immediately if the program is currently executing pure Java code; what matters is that no more macro commands are being executed). At the same time the MacroEngine is informed about the context switch. Shortly after that, the current context (which contains the now paused program) is swapped onto the swap stack and the a new context is created for the newly arrived program. As the program before it, this program is initialized with references to the proxy objects and then run within an own thread.

A Context contains a reference to the program that runs within it and a maximum of three references to handlers that might have to be activated in case of a sensitivity event. It also records all of the context’s state information.

A complicated protocol, and the Executor’s state machine (discussed in chapter 2) keep track of modes, states and execution levels and keeps the MacroEngine in sync.

**Implementation of the `breakMacroSeq` command**

A last detail must be explained here: how the `breakMacroSeq` command is implemented (this is a handler specific command that forces the interrupted program to continue after the end of the sensitivity block within which an interrupt occurred). As we’ve already stated several times, we can not (and neither wish to) influence the execution of the ”regular” Java commands, because we do not have control over the JVM’s execution from within Java itself. As a consequence, we can certainly not enforce another flow of control or make the JVM skip arbitrarily many statements. But how do we realize a command like `breakMacroSeq` given these circumstances?

The first issue that we must clarify here, is the exact semantics of the `breakMacroSeq` command. As the name already suggests, we are talking about skipping macros here - not about skipping the regular Java commands. This command is intended to be used by handlers that perform a certain sequence of commands to maneuver the robot out of an exceptional situation.

Typically, a robot starts off in a state $s_1$, performs a task $t$, at the end of which it ends up in a state $s_2$. Task $t$ is composed of a sequence of macro commands $c_1...c_n$ that define a path leading from $s_1$ to $s_2$. If an exception occurs during the execution of some command $c_k$, with $k < n$, a handler procedure is executed, which in turn is composed of commands $h_1...h_n$. Thus the robot finally executes the following trace: $c_1...c_k h_1...h_n c_{k+1}...c_n$. However, because the handler commands may define an alternative to some sub-path within the original trace, we do not wish to execute the corresponding commands of the original trace as well. In that case, our idea of the finally executed command sequence is more like the following one: $c_1...c_k h_1...h_n c_{k+p+1}...c_n$, with $p \geq 0$. Thus the handler commands $h_1...h_n$ effectively define an alternative to the original commands $c_{k+1}...c_{k+p}$, if $p > 0^2$. Once the handler procedure has completed, we

\[2\] If $p = 0$, we do not really skip anything but rather continue at the place where we were interrupted. In that case, the handler procedure commands do not provide an alternative,
consequently continue the original program at $c_{k+p+1}$.

What we wish for the `breakMacroSeq` command to do is to skip the commands $c_{k+1}$ to $c_{k+p}$, if $p > 0$. However, intertwined with the macro commands that are to be skipped, we have arbitrarily many regular Java commands. We strongly assume that those commands are typically used for maintaining an internal state of the robot. No matter whether we execute a handler or not, that state must be the same at $c_{k+p+1}$, because any alternative sub-path $h_1...h_n$ that is executed by a handler procedure will join with the original command trace here. As a consequence we do not need to prevent the execution of the regular Java commands, because they just describe a transition from state $s_1$ to $s_2$, assuming that $s_1$ is the state which is valid at $c_1$ and $s_2$ is the state valid at $c_n$. Since the identical transformation would have to be performed by the handler, in order not to corrupt the internal state, we can spare us of that and just leave it to the Java commands $j_1...j_n$ of the original trace.

Before going into the details on how we implement such a behavior, let us quickly summarize the exact semantics of the `breakMacroSeq` command within the Macrobot framework:

At the end of a handler procedure, consisting of commands $h_1...h_n$ and starting after some macro command $c_k$ within a sensitivity block, the original macro command sequence $c_{k+1}...c_{k+p}$, interspersed with arbitrarily many Java statements $j_1...j_n$, is replaced by those Java statements only and then continued at $c_{k+p+1}$, where $c_{k+p+1}$ corresponds to the `endSensitive` command closing the sensitivity block that was defined with respect to the sensor from which the sensitivity interrupt originated.

The Executor implements those semantics with a conceptionally simple trick: if a `breakMacroSeq` command is encountered at the end of a handler procedure, the Executor first flushes the rest of the currently halted macro on the MacroEngine. It then redirects the communication with the Spine to a `Nil` device and continues the macro program at the place where it was interrupted. The `Nil` device acts as a proxy for the MacroEngine, and pretends the execution of the command requests that are sent to it. I.e., it ignores all requests and simply returns ack-messages for them as if they had been executed. Just before the closing `endSensitive` command is executed, communication with the real Spine is re-established and the program proceeds as if nothing had happened.

Thus the execution of the handler procedure is rendered completely transparent to a macro program, which is exactly what we want.

### 3.6.2 MacroEngine

The implementation of the MacroEngine and its associated components is depicted in figure 3.13.

The MacroEngine is working similar to the Executor that acts as its client. Requests from the Executor modules arrive through the CommManager. Like the Executor, the MacroEngine also has different states of execution. However, the state machine that we implemented here is much simpler than the one for the Executor. Basically it only keeps track of the current execution level, that is, one

---

but an extension to the original trace.
of basic, task or handler. State changes are driven by explicit synchronization requests (begin task, task done and handler done) initiated by the Executor as well as by the detection of sensitivity events (equivalent to a begin handler request).

The beginning and end of basic programs are not communicated because the machine starts in the basic state and the execution model specifies that a basic program never ends (it can only be temporarily swapped out in favor of another program).

The message protocol between Executor and MacroEngine specifies that there can never be two execution requests overlapping in time. Normally, the first request must always terminate before a second execution request can arrive, with one exception: if a synchronization message (i.e., a context/state change request) arrives asynchronously while a macro is still running, then that macro will be paused and swapped out. The engine is then ready to receive another execution request (issued by a program with higher precedence), although the macro which has been halted has not finished yet, technically. Going into handler mode does not only enforce abortion and swap-out of the currently running macro, but also stops the SensorWatch from producing further sensitivity events until a handler done message arrives. This is a consequence of the execution model, which specifies that no sensitivity events must be processed during the execution of a handler procedure.

Every macro execution request contains the ID of the macro to be executed along with parameters for the macro. Once the request packet has been decoded, the MacroEngine fetches a (singleton!) instance of the Macro with the corresponding ID from the MacroFactory and then calls its execute method with the given parameter(s). If it completes with 100 percent, the execution of the macro command is acked and the engine waits for the next request to arrive. If the macro was aborted (which can only happen in case of an asynchronously requested state change) then its execute method returns with less than a 100 percent. In that case the execution request is not acked until the macro is either completed or flushed lateron.

Due to economical reasons, the MacroEngine was implemented with a some-
3.6 MacroMode

what reduced design, trading clean OO paradigms for efficiency to some degree. Swapper and context information, for example, are not modelled as classes of their own, but are implemented directly within the MacroEngine instead.

3.6.3 MacroMode Example

A sequence diagram illustrates best how exactly a macro program is executed. Figure 3.14 demonstrates the control flow among the MacroMode components during the execution of the example program listed in section 3.1.3.

![Sequence Diagram]

Figure 3.14: Executing a program in MacroMode

For an interpretation of the sequence diagram it might be helpful to know that the following components are running as individual threads: MacroEngine, SensorWatch, BrainController, Executor, Basic, and Handler. Most of the message passing among those modules is performed by setting flags, which are regularly inspected. The diagram does not include the communication stack, because it is not of importance here. It should be clear though that the messages
that are passed back and forth between the two proxies and the MacroEngine are actually sent as byte packets from Brain to Spine across the IR link.

### 3.6.4 Controlling MacroMode

By default, `MacroMode` is turned off upon start of the BrainController. Otherwise `MacroMode` is controlled in a similar fashion to `DirectMode`, see section 3.5.2.

### 3.7 Robot Prototype

The `Macrobot` framework is of no use, if we don’t have any robots that can be controlled with it. Building a robot for the `Macrobot` environment does not only comprise of assembling the actual physical construction but also of the implementation of "driver software", that is, the programming of concrete macro instances.

#### 3.7.1 Building a physical Robot

The time needed for the conception and actual building of a physical robot prototype should not be underestimated. Getting familiar with the workings of the LEGO mechanics takes its time.

As we've already stated in section 2.8, the realization of a robot that will be employed within the `Macrobot` framework introduces additional requirements for the architecture. Such an architecture must not only integrate the RCX module but also an iPAQ (with sleeve) in such a way that stability and maneuverability of the machine are not impaired. On top of that, the control panels of both components must be easily accessible for maintenance and control reasons.

Integration of three sensors must be performed in such a way that the robot model is optimized for usage within an enterprise setting, supporting control of the device by means of macro programs. I.e., sensors should be placed so that their events can be interpreted meaningfully and so that effective handler actions can be programmed that respond to the generated events in a meaningful way.

We have come up with a prototype robot model that fulfills all of those requirements. Its architecture is illustrated in detail in appendix D.

#### 3.7.2 Writing Macros for a new Architecture

Appendix C gives hints and guidelines concerning the implementation of architecture-aware macro commands. It also describes the steps that must be taken if the `Macrobot` system is to be extended with further macro commands.
Chapter 4

Debugging and Testing

This chapter describes the methods and strategies that were used for debugging and testing the Macrobot framework. We’ve dedicated an own chapter to this subject, because the development of our cross-platform system turned out to be challenging with many respects. Especially the testing of the RCX applications proved to be non-trivial, due to their embedded nature and the very limited ways of generating output.

4.1 Debugging RCX Applications

The RCX brick has only very limited ways of providing feedback for the debugging of programs. Compared to a workstation, which offers many different ways of logging program activities (e.g., logfiles or screen messages), the RCX hardly provides any means of recording or displaying the state of a running application.

4.1.1 LCD and Sound

When testing and running applications on the RCX, the programmer has to rely on visual feedback provided by the 5-digit LCD display and on audio signals produced by using the RCX’s sound capabilities. Both of those methods are only suited for debugging very specific characteristics of a program. Examples are the entering of a specific state or the change of such or maybe the computation of some result value. The main problem of communicating by means of the LCD and sound device is the very limited level of verboseness that can be produced. Every condition or statement that we want to make available to an external observer first has to be encoded into a 5-digit number or a sound by the application and then be decoded or interpreted in real-time by the tester. This brings us to the next big problem: we can not record our output in any way for later revision. In other words, we do not have the chance to debug our programs step by step. If we have frequently updated values that we wish to control or states that are rapidly changing, we will simply be lost by the many almost concurrent outputs that we would have to process both visually and auditive. This is, of course, especially difficult for multi-threaded applications where debugging output is often produced simultaneously. Here, a human tester does not stand a chance to follow the execution of a program. One will simply
not have a clue about what is going on.

There are several other disadvantages when using LCD and sound for debugging our applications:

Theoretically, because we are using Java, which is a platform independent language, all applications that run under LeJOS should also execute on a regular JVM. Hence, it should in principle be possible to test programs or components on a PC before they are used on the RCX. However, this is only true if an application or component does not interface with the RCX hardware, which is unlikely for a LeJOS program. If we debug programs by using the Sound or LCD class, we introduce extra RCX hardware dependencies that make it impossible to test applications on the PC which were previously hardware-independent\footnote{The opposite is true, too. If we have a truly hardware independent component, we may first test it on the PC using classical debugging methodologies, such as screen output for example. However, once we have cluttered our code with print statements, we will no longer be able to run it on the RCX, because LeJOS does neither provide a stdout nor a stderr stream to write to. Thus, we must remove all print statements after testing or comment them out. Conditional debugging (steered with a boolean variable, for example) is not a solution here, because it does not remove the reference to the non-existing system resource. An apparent solution would be the usage of a loosely coupled logger, specified through an interface class. Without too much trouble we could provide a different implementation of the logger component, depending on the platform we are compiling for (e.g., one that redirects logging output to stdout on the PC and another one that generates either LCD output or no output at all on the RCX).

However, that solution is a deceptive one. Although it really gives us the chance of not having to change our code with respect to the output statements, it just provokes trouble in another place: If our code is crammed with print statements, we deliberately waste precious memory resources for the instantiation of string constants (which are typically large due to their verboseness).}

A final drawback that results from the usage of Sound and LCD, is that utilizing any services of those LeJOS classes will cause their inclusion into the linked binary produced for upload to the RCX. This inevitably consumes several hundred extra bytes of precious memory, which we may not be willing to grant for usage. Note that this objection can only be raised if the LCD and/or the Sound class are not used for any other reasons than debugging. If the contrary should be the case though, they will be linked anyway and our argument becomes invalid, with respect to those two classes.

But even so, the point we wish to make remains an important one: if we are operating closely at the resource limits of the RCX, we don’t want or simply can not include any extra classes for debugging, because we just can’t afford to spare any memory for them!

\subsection{The \texttt{emu-lejosrun} Tool}

LeJOS comes with an own simulator tool, which is called \texttt{emu-lejosrun} and which allows for running the LeJOS environment on a workstation. Running a LeJOS application using this tool produces, for some cases, a more verbose output than if it were run on the RCX, especially when errors are occurring. Also, ROM calls are listed on screen which gives the user at least some idea of what is going during the execution of a program (provided, that the program interfaces with the ROM of the RCX, for example by calling to the RCX hardware). The simulator is, however, completely useless if one wishes to produce verbose output \emph{explicitly}, e.g., with print statements or the like.
As a matter of fact, *emu-lejosrun* is mainly of use if one must track down the exact location within code where an exception was thrown. Normally, if an uncaught exception occurs during the execution of a program on the RCX, LeJOS exits the application, beeps and displays both a 4-digit method signature ID and the exception class ID modulo 10 on the LCD display. The method signature corresponds to the method which threw the exception. When running the same program, *emu-lejosrun* will list the method signature ID as well, but additionally the complete exception ID will be provided, along with the ID of the root method (i.e., the method which called the guilty one), the thread ID and the bytecode offset of the faulty statement. Using the latter in combination with the standard *javap* command (the java file disassembler), we can find out the exact statement which caused the error.

To run any LeJOS applications within the simulator, we must link them first with *emu-lejos* (instead of *lejos*) and then run them with *emu-lejosrun* (instead of *lejosrun*). The option *-verbose* used with the linker produces a table of class and method signature IDs. The option *-v* used with the simulator leads to a better human-readable output of the simulator.

The LeJOS simulator tool has a major drawback: it does not simulate hardware input. In other words, if we poll sensitive hardware such as sensors or buttons, we will never get any sensible input values. There is also no way to provide *emu-lejos* with pre-fabricated or just-in-time values for those components. It gets even worse: if the application that we wish to simulate contains code that waits for a specific input (e.g., for a button to be pressed) we will end up dead-locked, thus such statements must be removed before running a program within the simulator.

Because most of the application code that runs on the Spine is dependent on either sensor- or infrared input (which can’t be simulated either for obvious reasons), we can draw the following conclusion: The *emu-lejosrun* tool is of little or no use at all for testing or debugging components within the Macrobot framework.

### 4.1.3 The Macrobot Simulator

Due to the various problems that we’ve described in the previous two sections, debugging the Spine system demanded a great amount of time and work during the development of the Macrobot framework. With increasing time, the need for a solution allowing to run the whole framework on a workstation grew steadily.

With the implementation of an own simulator we eventually arrived at such a solution, which fulfills all of our needs, in particular the following ones:

- run the whole framework on a workstation
- support rapid prototyping
- test and use the comm stacks without using the physical IR link
- produce verbose output (i.e., log to file or screen) from LeJOS applications
- generate sensor input at will, randomly or by hand
- display the (simulated) state of hardware components graphically
Rapid prototyping is supported by the Simulator, because it allows for a fast code-compile-run cycle, sparing us of lengthy upload times. We can even use a debugger, if we run the simulated framework!

Section 4.2 explains in more detail how the simulator was implemented. The remainder of this section only summarizes its usage.

Start and Usage of the Simulator

Running the framework with the simulator only requires the following preparation steps: first, include both brain.jar and spine-sim.jar on the classpath and second, run ch.ethz.inf.macrobot.Simulator instead of BrainController. That should do the trick.

Simulator actually extends BrainController, and overrides its initialization method. As a consequence, the functionality and public interface of the simulated Macrobot system are exactly the same as if the system were started with BrainController.

Using the simulator classes, we can now use print statements or various loggers (see ch.ethz.inf.macrobot.util package) to produce debugging output. Generally, running the framework with the simulator allows us to use any regular JDK API methods within the Spine classes. Of course we should consider the arguments that were given in section 4.1.1 and we should not forget that the Simulator was implemented mainly to give us the chance to perform rapid prototyping of the Spine classes within the whole Macrobot framework.

Although the Simulator gives us access to all the features of a regular JVM, we should always be aware that, in the end, most of that functionality will not be available under LeJOS. Thus, all the added debugging code should be removed once that the components are running fine. This is recommended practice not only because of the differences between Java and LeJOS, but also because it saves resources on the RCX. Unfortunately we cannot afford the luxury of having unused code within our classes.

The Simulator View

If we start the Macrobot system in simulated mode, the simulator view (shown in 4.1) will pop up. It displays individual views of all of the hardware components that are simulated (buttons or sound are currently not supported and thus not part of the view). On the left hand side, we see views that show the state of the battery, the LCD display and the three motors, on the right hand side, we have views that represent the state of the three sensors.

There are three types of hardware: components that we can manipulate manually only, components that we can manipulate through LeJOS only and components that we can manipulate both manually and with LeJOS. Table 4.1 groups the various hardware parts according to that classification (Sound and Buttons are only listed for the sake of completeness).

<table>
<thead>
<tr>
<th>LeJOS</th>
<th>Manual</th>
<th>LeJOS/Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motors, LCD, Sound</td>
<td>Battery, Buttons</td>
<td>Sensors</td>
</tr>
</tbody>
</table>

Table 4.1: Different hardware types according to manipulation interfaces
Figure 4.1: The interactive simulator view

The state of all of the components which we can manipulate by hand and which are also part of the simulator view can be edited with the mouse. The battery slider, for example, can be dragged to the right or to the left, increasing or decreasing the voltage of the batteries accordingly. The next section explains how we can use the simulator view to generate sensor events while a Macrobot is running.

The components that cannot be manipulated externally just display their internal state. If we change that state via LeJOS, the displayed value changes simultaneously.

Generating Sensor Values

The simulator view allows for controlling and producing raw sensor events. There are two ways to achieve this: we can either produce a single event of a given strength or we can continuously generate events over a period of time. In any case, the percentage value which is depicted in the currentValue field is the value which is currently read by the sensor.

The slider at the bottom of each sensor view defines the strength of the next sensor event to be generated. Moving the slider to the right or to the left, will not produce any values in normal mode. But if we press the generate button, an event of the specified strength will be generated, just as long as we hold the button down.

Sometimes we would wish a sensor value to change over time. Unfortunately, we can not move the slider and press the generate button at the same time. This problem is resolved by activating the "keep" mode of a sensor: if the keep checkbox is checked, the view behaves as if the generate button was pressed all the time. Events are constantly produced now, and if we move the slider up or down we can see how the sensor value changes simultaneously.

Please note that we produce sensor readings in the range [0..1023] by moving the nextValue slider. The currentValue field, however, displays the currently read value in percent. Since raw value to percent value conversion is not performed at an exact 10:1 ratio, it should not be considered an error if the currently read value equals 58 percent, although the slider specifies to generate a raw value of 601!
Limitations

The current simulator implementation has been specifically tailored for the needs of the Macrobot framework. The hardware which is not accessed by the system is thus (currently) not supported by the simulator. This mainly concerns the Sound and Button classes. They are not simulated, because there was no need for them.

The (simulated) communication link between PC and RCX is assumed to work flawlessly, we cannot influence the transmission quality.

Not everything can be realized with the simulator. There are inherent limitations that stem from the way that it has been implemented and designed. Because it does not really simulate the complete RCX hardware, we have no way of simulating the ROM or the Memory classes. We could of course provide empty implementations for those, so that they could be accessed on the PC as well, but they would not truly simulate their physical counterparts.

As a consequence, the Macrobot simulator does not really care about the memory constraints of the real RCX. Instead the whole memory of the JVM is at disposal. No limitations, with this respect, are imposed on the applications that are run within the simulator.

4.1.4 JUnit Tests

JUnit is an open source Java testing framework used to write and run repeatable tests. It is intended for writing unit tests and is thus only of limited use for testing the Macrobot components.

An unit is a software module or set of modules that exports a variety of services which can be utilized by other software modules. Typically a unit produces results which are either returned by its service methods directly or which can be measured otherwise. Library components are a good example for units.

A JUnit test is implemented as an (all-knowing) client of the unit to test. The programmer writes a test that calls upon all the services of the component with various parameters. The test program already knows the correct result in advance, and compares the result of the call with the result that it was expecting. If there are discrepancies, the test fails. The JUnit libraries offer a series of assertions that can be used to test for the presence or absence of certain conditions after the call to an unit method has returned.

With JUnit one can easily build whole test suites that can be extended incrementally with new tests or test suites for newly added components. Thus a test suite for a software system ideally grows in parallel with the system that is being developed. Because the tests are automated, they can be run at any time without much effort.

But JUnit is not well suited for testing the components of the Macrobot framework for several reasons:

The main reason is that we hardly employ any units, in the sense as described above, within the framework. Most of our modules do not produce deterministic results, i.e., the results that are returned are based on arbitrarily produced values (e.g., the value of a sensor, which can hardly be foreseen). If we wanted to make those results deterministic (and thus suitable for being tested with a unit framework) we would have to implement a simulation process which
(re-)produces the same values at all times. This has not been done due to the lack of time, but also because it is not the goal of unit tests that one must write a whole back-end structure for the components that one wishes to test.

Also, Macrobot is a very tightly coupled system. I.e., components could hardly be tested in an isolated fashion. Most of the system had to be completed before one could even think about unit tests, and in the end there were only a few candidates left for unit testing (namely BrainController for testing the whole system, RcxProxy and Executor for testing DirectMode and MacroMode, respectively). The methods that are exported by those, however, do not produce results that can be tested. How would we test whether a robot actually turned on its motors and actually drives about a certain distance in the required direction, or if the sensor value that was returned was the correct one?

Summing things up, we can see that the very nature of the Macrobot system denies the usage of unit tests. However, there are a few components that were suited to be tested with JUnit. We took advantage of this testing method where it was appropriate. The comm stack layers, for example, were tested in this way.

4.1.5 Interactive Tests using DirectMode

For some special cases, DirectMode is an ideal tool for debugging and testing the functionality of the RCX hardware. For example, we can use DirectMode to get an idea about the amount of free memory that is currently available on the RCX. This information helps a great deal in deciding about what features we should additionally implement and how we should do it, as well as to get a general idea of how certain changes in the system affect memory usage.

Because we can write to and read from most of the RCX’s hardware components by using it, DirectMode is very well suited to quickly test the hardware of a robot or to get an overview of the workings of certain parts.

The DirectMode Shell, an application which allows to access and utilize the DirectMode API interactively, is of particular use for such tests. Its usage is described in more detail in appendix B.

4.2 The LeJOS Simulator

To aid the process of rapid prototyping and to help debugging and testing RCX applications we have eventually implemented a simulator which allows to run LeJOS programs on a workstation. The result of this side project simulates only the most important hardware, because it was conceived out of despair for a solution to the RCX debugging problem and there was no need for a more sophisticated solution. However, the concept of the current simulator makes the implementation of a more complete version very practicable.

4.2.1 Design

The concept that stands behind our simulator is a very simple one: the simulator is simply realized with reimplementations of all the LeJOS API components that interface with the hardware.

The josx.platform.rcx package contains several classes that provide an abstraction of the various LEGO MINDSTORMS hardware, Motor, Sensor or
Battery, for example. Most of the Java classes within the rcx package are empty interface classes, which are implemented by corresponding native libraries.

By providing new non-native implementations of those classes that simulate the underlying hardware, we can detach our applications from the RCX and resolve the hardware dependencies. To run such an application under Java instead of LeJOS, we must simply include a new release of the LeJOS libraries, which contain the simulator classes in place of the original native classes, in the classpath of the application. Figure 4.2 illustrates the idea of the simulator.

![Conceptual design of the Macrobot simulator](image)

**Figure 4.2:** Conceptual design of the Macrobot simulator

### 4.2.2 Implementation

The term Macrobot simulator stands for a set of classes that are scattered across the whole framework and which, in combination, essentially form and bring about the simulation effect. When the Macrobot framework is run in simulated mode, the classes of the simulator replace some of the regularly used classes.

#### Hardware Classes

The Macrobot simulator replaces some classes of the josx.platform.rcx package with alternative implementations. Those implementations are programmed according to the MVC (Model-View-Controller) pattern. I.e., they all consist of a **model** that effectively simulates the state of a hardware component, of an (optional) **control** module, which allows to change the state of the model interactively by using mouse and keyboard, and of a **view** which graphically represents the state of the model.

The current implementation of the Macrobot simulator replaces the following LeJOS classes with simulator components: **Sensor**, **Battery**, **MinLCD** and **Motor**.

#### Communication Link

Simulation of the communication link is another central feature of the Macrobot simulator.

The implementation allows to simulate the comm stack to various degrees. This is achieved by cutting off the lower part of the Janus protocol stack at various levels below the CommManager, and by linking the two remaining stubs with two interconnected **PacketHandlerProxy** instances. This works just fine, because both the CommManager and the duplex layer of the protocol stack work with an abstract PacketHandler on their lower end.
4.2 The LeJOS Simulator

We also have the choice to bypass the Janus stack completely, and to use two directly connected PeerCommManager components for the communication between Brain and Spine.

Figure 4.3 shows the various ways of simulating the comm stack. A Factory class, called CommStackFactory, is used to create all of the depicted variants and to link them correctly.

The Simulator Class

The Simulator class is a sub-class of the BrainController class and overrides the init method of the BrainController. It exports the exact same API as the regular BrainController and can thus be regarded as such by any client applications that wish to use the simulated framework.

Running the ch.ethz.inf.macrobot.Simulator class starts the simulated Macrobot system and initializes the various components, such as the comm link and the SpineController. First, an appropriate comm stack is created. Then, the SpineController is run within an own thread and hooked up to one end of the communication channel and finally the Simulator links itself to the other end. The rest of the initialization is performed by the BrainController and is similar to the start-up of the normal two-platform system.

4.2.3 Usage

Section 4.1.3 discusses the usage and the workings of the simulator in more detail.

4.2.4 Limitations and Extensions

Of course we do not provide a "real" simulation of the RCX environment with this design. We can, for example, not simulate the limited memory that we are actually confronted with on the RCX. But that was not the goal of the Macrobot simulator anyway. What we primarily wanted, was a way to run and test hardware-dependent LeJOS applications on a workstation. This goal has been achieved.

For the hardware that provides feedback we have implemented a visual control interface which gives a human user the chance to produce arbitrary
values at will. However, the MVC pattern makes it feasible to extend the simulator with alternative controller implementations that, for example, use a statistical process to produce values automatically. Using the same technique, it should also be possible to reproduce the same series of values at any time that we wish, for example by reading them from a file or by using a deterministic function over time that generates values from an initial seed value.

Implementing the latter would probably make it feasible to use unit tests (see section 4.1.4 above) for a greater set of components within the Macrobot framework, because we can then anticipate results that are based on hardware feedback.
Chapter 5

Challenges and Lessons learned

This chapter talks about the challenges that we’ve encountered during the deployment of the Macrobot system. Not all of them were mastered, but nonetheless there were many lessons to learn that may be of good use for anybody who may be continuing the work on the subject of this thesis.

5.1 iPAQ

The Compaq iPAQ handheld that we used for the prototype is an 3800 model running Familiar Linux. It is equipped with a "sleeve" extension module which holds a microdrive, a wireless LAN card and an extra battery pack. The iPAQ has an infrared port and a proprietary multi-port interface, the latter of which supports USB and serial connections with a PC for synchronization purposes as well as the powering of the device with external power.

5.1.1 Cross-compiling

iPAQ handheld computers are shipped with an ARM processor. Therefore we cannot use any application binaries that were produced for a regular workstation. Instead we must rebuild all applications that we wish to employ from sources and cross-compile them for use on the ARM architecture of the iPAQ.

For the cross-compilation and porting of the LeJOS C-tools (the API classes, of course, must not be recompiled, because they are platform independent Java files) we employed the open-source skiff cross-compilation tool chain. It is available from [22], which also provides further information on the topic of cross-compilation for the ARM v4l (4=version, l=little endian) target.

Some of this general information, as well as details concerning the cross-compilation of the LeJOS tools, has been condensed in the following files (relative to the Macrobot project root):

docs/ipaq/CROSS_COMPILING_TOOLCHAIN_README
docs/ipaq/README
There is a script (scripts/lejos_crosscompile) which performs cross-compilation of the LeJOS tools and upload of the produced binaries to the ikpaq2 via secure copy. The script should be executed from the project’s root directory.

5.1.2 Communication with the RCX

Unfortunately we have encountered various hardware-related problems during the deployment of the Macrobot prototype, which have prevented us from establishing a connection with the RCX from an iPAQ (PC works fine). This section illustrates the means of communication that we investigated and the various difficulties that we faced.

Using the USB Port

There is no way of using the USB version of the LEGO IR-tower together with an iPAQ, although the handheld does offer USB support. But the iPAQ is internally configured as an USB client and not as a master device, hence it will never be able to make use of another USB client such as, for example, the USB IR-tower.

This is bad news mainly for the reason that all of the newer MINDSTORMS kits are equipped with USB towers only. However, extra serial towers can still be ordered from the manufacturer.

Using the serial Port

A serial sync cable is available as an extra asset for the iPAQ. By using such a cable together with a gender-changing clutch, we can hook up an iPAQ with a serial LEGO IR-tower.

For our prototype, we succeeded in utilizing the serial sync connection together with minicom for upload of files. However, the usage of a serial IR-tower (connected in the way as described above) together with the cross-compiled LeJOS tools (e.g., lejosrun or lejosfirmnd) failed for unknown reasons. Why exactly this isn’t working is still an unresolved issue. Since communication over the serial port works in general, we suppose that it might be a timing issue in the LeJOS tools, but the timeframe of this thesis hasn’t permitted to further investigate the problem. A few attempts have been undertaken to patch the tower library code but none of them succeeded.

Using the built-in infrared Port

The iPAQ has a built-in infrared port, which in principle can be used to communicate with the RCX, eliminating the need for the tower. This solution would be very much appreciated, since it removes the necessity of having to integrate the tower into a robot’s architecture and would thus reduce both weight and size of robot models. But we say "in principle", because the way that the RCX’s IR port works is different from that of the iPAQ’s IR port. This incompatibility is mainly due to the different frequencies that are employed by the individual ports.

We have already given thought to the IR issue at an early stage of this thesis, but no solutions were in sight back then. Many people had tried to realize a working system (which is documented in various internet discussion forums), but nobody succeeded. However, during the end-phase of the thesis, a new very
promising project, called RCX-IR [16] was started on Sourceforge. According to the project documentation, the developer managed to write a library that actually allows for communication with the RCX by using the iPAQ’s built-in IR port.

However, an appropriate Java wrapper (i.e., an alternative implementation of LeJOS’ Tower class) had not been written yet and there was no support for the LeJOS tools so far. Since we did neither have the experience and knowledge nor the time for realizing those missing pieces ourselves we never got a chance to test the Macrobot system on top of an RCX-IR communication. But we did succeed in installing the latest release of RCX-IR on the iPAQ together with the required kernel modules. Therefore, all the necessary prerequisites for a later usage of the system are given.

If the RCX-IR project proves to be a success, it will quickly receive support from the LeJOS developing team, we are positive of that. Once that LeJOS will be running satisfactory on top of the new communication link, we can then subsequently include it into the Macrobot framework.

5.2 LeJOS

LeJOS 2.0 is the result of more than two years of active development by a small team of open source developers and has emerged as a very stable alternative to the original Lego firmware. Generally speaking, there are almost no known bugs in the current version, and if there appear any, they are usually fixed within a few days. The development of the system currently focuses mainly on enhancements of the API.

5.2.1 Patching of existing API

The LeJOS 2.0 Thread implementation doesn’t support the Runnable interface. Thus ”multiple inheritance” from both Thread and any other class is prevented. For the sake of being able to run any class that implements the Runnable interface within an own thread, we have patched the sources of LeJOS 2.0 for support of this feature within the Macrobot system.

5.2.2 Integration into the Macrobot Framework

LeJOS is an integrative part of the Macrobot framework. Hence we included all of the sources of version 2.0 in the Macrobot project. This solution allows for building LeJOS anew on any platform that employs Macrobot. It also guarantees that LeJOS will be available on any system that runs the framework. The local (i.e., local with respect to the project) patching of any LeJOS classes - be it due to bugs or due to missing functionality - becomes very feasible with this solution.

However, we did not want to add the LeJOS CVS resources ”as is” to our project. Although such an integration might have been very convenient, considering updates or bugfixes, we would have also added a lot of needless beta code. Since we wish to keep the sources as clean as possible in order to facilitate a quick overview of the LeJOS system, the original CVS LeJOS project structure was thinned out and refactored for integration into the Macrobot project.
Makefiles have been adapted or newly created where it was necessary. The following list gives a synopsis of the contents of the \texttt{lejos} subdirectory located in the root of the \textit{Macrobot} project:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin/</td>
<td>LeJOS tools, LeJOS binary (.srec) and irtowerlib</td>
</tr>
<tr>
<td>classes/</td>
<td>LeJOS 2.0 sources</td>
</tr>
<tr>
<td>classes/README</td>
<td>Explains the source directory structure</td>
</tr>
<tr>
<td>common/</td>
<td>Database information needed for linking</td>
</tr>
<tr>
<td>doc/</td>
<td>Various documentation and LeJOS2.0 javadocs</td>
</tr>
<tr>
<td>jtools/</td>
<td>Sources for the Java-based LeJOS tools</td>
</tr>
<tr>
<td>lib/</td>
<td>Jar-files with LeJOS2.0 classes</td>
</tr>
<tr>
<td>tools/</td>
<td>Sources for the C-based LeJOS tools</td>
</tr>
<tr>
<td>tower/</td>
<td>Sources for the IR-tower library</td>
</tr>
<tr>
<td>vmsrc/</td>
<td>Sources for the LeJOS VM</td>
</tr>
<tr>
<td>README</td>
<td>Information about LeJOS as part of \textit{Macrobot}</td>
</tr>
<tr>
<td>INFO</td>
<td>Information about the directory structure</td>
</tr>
<tr>
<td>Makefile</td>
<td>Builds LeJOS under \textit{Macrobot}</td>
</tr>
<tr>
<td>cctest.sh</td>
<td>Compiler configuration script</td>
</tr>
</tbody>
</table>

The new structure makes it necessary to copy any new or altered source files or bugfixes by hand to the respective new location. One should \textit{only import source files, never} any classfiles from a new distribution. The distribution class archives and the LeJOS tools in the \texttt{lib} and \texttt{bin} directories, respectively, should always be built from scratch (using make) if the sources have been altered.

### 5.2.3 Further Advice

There are several resources for getting advice and information on various topics concerning LeJOS:

**lugnet.robots.rcx.java** is a discussion group served by lugnet (Lego User Group Network), concerned with programming of the RCX in Java, i.e., with LeJOS. The group is not very active (in contrast to the super group, which is concerned with programming of the RCX by any means), but might still be a helpful source for information about various issues concerning LeJOS or for discussing problems.

**LeJOS discussion** is the official LeJOS mailing list. It is a very rich source of information and we suppose that about 90 percent of all issues concerning LeJOS are discussed here. The list is moderated and frequently posts are answered by someone of the LeJOS development team personally. Any announcements about new LeJOS releases, bugfixes or other developmental issues are exclusively published in this source.

**www.lejos.org** is the official home page of the LeJOS project. Apart from links to the CVS sources, a download area and general information about LeJOS there is also a small Wiki-FAQ maintained by users. However, the FAQ entries are sparse, and for answers to frequently answered questions one better consults the archives of the LeJOS discussion mailing list.
5.3 Working with limited Memory

The RCX offers a total of 32 kB of memory, 17 kB of which are occupied by the LeJOS firmware, an additional 4 kB is not available due to various reasons. This leaves us with 11 kB of freely usable program space, which must suffice to hold program code, execution stack and any allocated objects. The binary program file that is produced by the LeJOS linker is copied verbatim into the memory by the LeJOS loader on the RCX, as soon as data is conceived over the IR link. Hence the effective program size is given by the size of the binary, plus an extra amount of memory needed for the allocation of stack and heap variables, which cannot be exactly determined.

11 kB of memory are not much and quite a challenge for programming sophisticated applications within those limitations. This section describes our own experiences with limited memory and gives a few hints and tips on the subject of economical memory usage.

5.3.1 Programming without GC

The missing garbage collection under LeJOS is not really a problem, if one is aware of it. What it means, though, is that we must think very carefully about where and how we instantiate new objects and that we should know at all times how many objects we have instantiated. Because the heap space, which is allocated by object instantiation, will never be freed again, we should either use the singleton pattern wherever possible or otherwise reuse all objects that we’ve once created. Instantiating objects "on the fly" is tabu and must not be practiced, as the references to those objects, and thus the memory that they occupy, will be lost forever.

Programming without GC also means that we should know the workings of Java, with respect to object instantiation. The frequent usage of Strings is a pitfall in an object oriented environment without GC, because each time when a String is used there will be memory allocated for it on the heap which is never returned. We are normally not aware of this fact, because the instantiation of Strings is comfortably performed with double quotes and not with an explicit new command.

The following list of rules concerning object usage should be followed as a minimum when working with LeJOS:

- Never instantiate objects or arrays for local use only.
- Do never create objects within a loop.
- Don’t use strings on the fly, reuse messages. Better even: don’t use strings at all.
- Instantiate any exceptions that are thrown statically. Thereafter rethrow the same instance all the time.
- Reuse objects once you’ve created them, i.e., instantiate objects only globally.
- Initialize arrays of objects member by member as needed.
• Generally try to instantiate lazily, i.e., do not create objects before they are really needed.

This list is certainly not an exhaustive one, but it illustrates the general strategy one must adhere to.

5.3.2 Using simple Types

An integer variable consumes four bytes of memory, a double eight bytes. However, often we use only numbers within a small range, and the use of a byte or short variable would be sufficient. By consequently using bytes or shorts we can literally save heaps of memory, especially in the case of global constants or variables.

However, we must be careful when performing calculations with byte values. In general we have the impression that a byte holds a value in the range 0..255, but this is not true in Java, where bytes are a signed type and actually hold values in the range of -128..127. All arithmetics in Java are typically done with integers, which may produce strange behavior when accessing byte variables that contain values within the range 128..255. The reason for this is, that, because of the signed byte type, a value of 254 will be read as -2 if used in a calculation or if it is assigned to another signed variable type, e.g. integer. We can receive the original value by masking it out with $0xff$, but this conversion is very tiresome and leads to error prone and unreadable code. A better practice may be to use short type variables if one is dealing with relatively small values that are greater than 127.

5.3.3 Reducing Design

Generally, memory can be traded for design and vice versa. But reducing OO design is normally not a good idea, because it helps keeping things in order and leads to well readable code. This is almost always worth much more than the few bytes we can save by degrading the level of design.

5.3.4 New LeJOS Linker

The LeJOS linker generates a binary file which is then uploaded to the RCX and copied into memory verbatim. Generally, the linker creates the transient closure over all type references and then includes all of the used classes into the binary. However, often we use not more but a single method of a class and the linker still includes the code of all other methods that are part of that class.

After a discussion of that subject on the LeJOS mailing list, the release of a new linker has brought great advantages in memory saving terms. This because the new implementation actually detects all of the methods that are used by an application and then eliminates the code of the unused methods from the binary. The new linker reportedly achieves savings in the size of the binary file between 10% and 25%, each byte of which is actually saved on the RCX.
Chapter 6

Conclusion

This chapter analyzes the results that have been produced by this thesis and draws some conclusions. It also gives suggestions for future work based on the Macrobot system.

6.1 Results

Framework

The Macrobot framework has been implemented according to the specifications of the assignment. For the communication between Brain and Spine we implemented an own full-duplex protocol that allows for reliable asynchronous information transfer between PC/iPAQ and RCX. Apart from the implementation of the central framework feature (i.e., to provide means for controlling LEGO MINDSTORMS robots by running abstract, architecture-independent macro programs), we have also implemented an extra controlling mechanism, called DirectMode, which allows for testing and steering a specific robot architecture on a low-level basis.

Simulator

We encountered numerous difficulties in testing and debugging the Spine parts of the framework (which run on the RCX). That fact lead to the conception and implementation of a simulator that allows to run, debug and thoroughly test LeJOS programs on a workstation, before they are migrated to the RCX. The simulator provides access to simulated RCX hardware and allows the user to produce and graphically view events on that very hardware. A simulation of the infrared communication link is also a central feature of the Macrobot simulator.

Prototype

In order to test the frameworks functionality we also built a physical robot model which integrates both RCX and iPAQ. For this prototype we implemented concrete drivers for the four basic macros forward, backward, turnLeft and turnRight that can be used to steer the robot.
Applications

A number of helpful applications that make use of the features of the Macrobot system have been written and are ready for usage. They are well suited for demonstrating the capabilities of the system and support rapid prototyping by providing interactive environments for the steering of individual robots.

6.2 Evaluation

The results from the last section give proof that the targets that were defined by the initial task outline of this thesis have been reached. Although the adaptivity and networking layers were not realized due to time constraints, we are convinced that the realization of the scenario that we’ve sketched in the motivating paragraph of chapter 1 would be possible with the current system\textsuperscript{1}.

6.3 Limitations

The current system also exposes a major limitation: since we didn’t succeed in realizing a working communication between the iPAQ and the RCX the actual system only supports the scenario where the Brain is located on a PC, communicating with the Spine on the RCX by means of an IR-tower. This condition prevents us from writing and executing programs for completely autonomous robots. There will literally be strings attached to any actual experiments, because the IR-tower must remain connected to the workstation.

The most promising solution to the communication dilemma lies within the results that the future development of the RCX-IR project may produce. Once that LeJOS can use the built-in infrared port of an iPAQ or any Laptop computer, we are able to eliminate the need of the IR-tower completely.

Another solution would be that of wiring up an iPAQ and an RCX directly, i.e., by providing a serial connection that actually bypasses the infrared communication. This solution was proposed only recently and hasn’t been investigated thoroughly yet. Although it also eliminates the tower from our scenario, it also introduces some new issues. We would have to implement a serial driver for the LeJOS classes on the Brain side, one that doesn’t care about the peculiarities of the serial tower. Also, because we couldn’t make the LeJOS tools and the Tower implementation work on an iPAQ with an attached serial tower, we are not sure of whether this seemingly more fundamental problem would be resolved with a direct serial communication.

6.4 Future Development

First target for any continuing work on the Macrobot framework must be the completion and implementation of a working communication between iPAQ and RCX either by means of the RCX-IR library or by hooking up the two components directly with a serial cable, completely bypassing infrared communication.

\textsuperscript{1}One could, as a matter of fact, use the Explorer macrobot program for this purpose, an example of a basic behavior that was given in section 3.1.3.
Further future development of the system should then mainly target the implementation and integration of the adaptivity layer and of the networking layer. For both tasks, components from existing IKS projects (Prose and Midas) can be used which should make their realization fairly uncomplicated.

After the addition of the adaptivity layer, experiments and research in the field of application-awareness in enterprise settings can be conducted. We are confident that the Macrobot system will live up to its expectations in and that it will well serve its purpose of supporting such investigations.

The realization of a standard setting with several robots, possibly of different nature, would then be the next logical step. It includes the implementation of intelligent room services that control the adaptation of robots within their reach in an automated fashion.

To facilitate the task of uploading new behaviors and programs to robots over the network, a general front-end application should be implemented. This application should also offer support for controlling each robot individually by means of DirectMode, possibly by integrating the DirectMode Shell, which is already part of the current system.
Appendix A

Project Structure

This appendix gives an overview of the Macrobot project structure, which is different from the standard IKS project structure.

A.1 Directories and Files

The following table reflects the directory structure of the project after a CVS checkout and describes the function and contents of the most important directories and files:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cross/lejos/</td>
<td>LeJOS cross-compiling aids.</td>
</tr>
<tr>
<td>docs/</td>
<td>Various documentation.</td>
</tr>
<tr>
<td>docs/ipaq/</td>
<td>iPAQ-specific documentation.</td>
</tr>
<tr>
<td>docs/test_output/</td>
<td>Output of various functionality tests.</td>
</tr>
<tr>
<td>docs/api/brain/</td>
<td>API documentation for the Brain subsystem.</td>
</tr>
<tr>
<td>docs/api/spine/</td>
<td>API documentation for the Spine subsystem.</td>
</tr>
<tr>
<td>lejos/</td>
<td>Partially extended and patched LeJOS 2.0 system (see 5.2.2 for more detailed information).</td>
</tr>
<tr>
<td>scripts/</td>
<td>Various scripts (see A.3) for ad-hoc network setup, application startup, etc.</td>
</tr>
<tr>
<td>src/MAIN.mf</td>
<td>Manifest for the brain.jar file.</td>
</tr>
<tr>
<td>src/lejos/sim/</td>
<td>Sources for the simulator.</td>
</tr>
<tr>
<td>src/macrobot/brain/</td>
<td>Sources for the Brain subsystem.</td>
</tr>
<tr>
<td>src/macrobot/spine/</td>
<td>Sources for the Spine subsystem.</td>
</tr>
<tr>
<td>src/macrobot/common/</td>
<td>Shared sources for both the Brain and Spine subsystem.</td>
</tr>
<tr>
<td>src/tests/junit/</td>
<td>Various JUnit tests.</td>
</tr>
<tr>
<td>BUILD_AND_RUN</td>
<td>General instructions on how to build and run the Macrobot system.</td>
</tr>
<tr>
<td>Makefile</td>
<td>Builds and runs the project (see A.2).</td>
</tr>
</tbody>
</table>

A.2 Makefile

The Makefile in the root directory of the Macrobot project should be used to build the project. It also offers various other services, such as starting the
Brain or uploading the linked binary to the RCX. The following table lists and discusses the various make targets:

<table>
<thead>
<tr>
<th>make Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brain</td>
<td>Build the Brain application.</td>
</tr>
<tr>
<td>spine</td>
<td>Build the Spine application.</td>
</tr>
<tr>
<td>simulator</td>
<td>Build the simulator.</td>
</tr>
<tr>
<td>tests</td>
<td>Build the tests.</td>
</tr>
<tr>
<td>lejos</td>
<td>Build LeJOS.</td>
</tr>
<tr>
<td>upload</td>
<td>Upload Spine to the RCX.</td>
</tr>
<tr>
<td>runbrain</td>
<td>Run the Brain application.</td>
</tr>
<tr>
<td>runtests</td>
<td>Run the JUnit tests.</td>
</tr>
<tr>
<td>javadoc</td>
<td>Generate Javadoc API document.</td>
</tr>
</tbody>
</table>

A.3 Scripts

Various scripts are available from the script directory of the project. The following table lists the exact purpose for each of them:

<table>
<thead>
<tr>
<th>Script Name</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipaq_comm_setup</td>
<td>iknlab_id, ikpaq_id, iface</td>
<td>Set up an ad-hoc communication with an ikpaq and configure the wireless LAN. Run without arguments for help.</td>
</tr>
<tr>
<td>ipaq_login</td>
<td>ikpaq_id</td>
<td>Log in to an ikpaq.</td>
</tr>
<tr>
<td>lejos_crosscompile</td>
<td>-</td>
<td>Cross-compiles LeJOS and uploads all essential files to ikpaq2.</td>
</tr>
<tr>
<td>smonitor</td>
<td>-simulate</td>
<td>Start the SensorMonitor.</td>
</tr>
<tr>
<td>dmshell</td>
<td>-simulate</td>
<td>Start the DirectMode Shell.</td>
</tr>
<tr>
<td>mmshell</td>
<td>-simulate</td>
<td>Start the MacroMode Shell.</td>
</tr>
</tbody>
</table>
Appendix B

Applications

A number of applications that act as clients of the Macrobot system have been implemented during the deployment of the framework. All of them have emerged from test programs that were originally implemented to verify the functionality of a certain feature of the framework. This appendix describes them in a quick overview.

Note A source code analysis of the described applications gives a good illustration of the actual workings of the Macrobot framework and its services.

B.1 Sensor Monitor

The SensorMonitor (its main window is shown in figure B.1) is an example of a very simple application that is based on DirectMode. It continuously polls all three sensors of the RCX and displays their values graphically. With radio buttons, the user can set the mode of each sensor individually and thus determine the way how values are displayed. The program also shows the roundtrip time of request packets as "update interval time" at the bottom of the window.

The power and simplicity of DirectMode applications becomes evident when we have a quick look at the source code that effectively reads the sensor values:

```java
    time0 = System.currentTimeMillis();
    values = rcx.readSensors(modes[1], modes[2], modes[3]);
    freq = System.currentTimeMillis() - time0;
    frequency.setText(""+(freq/1000.0)+" sec");
    s1.addData(rcx.getValue(values, rcx.S1));
    s2.addData(rcx.getValue(values, rcx.S2));
    s3.addData(rcx.getValue(values, rcx.S3));
```

If we ignore the code pieces that are concerned with timing the roundtrip-time, we can see that the whole magic is performed by four lines of code only. Because the Rcx object is effectively a serializable proxy, this code stays exactly identical, no matter whether the application runs locally on the computer that runs the Brain’s JVM or whether it is making use of the services remotely, e.g., via Jini.
B.2 DirectMode Shell

The Sensor Monitor can be started with the ApplicationRunner class:

```
java ch.ethz.inf.macrobot.ApplicationRunner [-simulate] smon
```

The optional -simulate argument starts the SensorMonitor in simulated mode, i.e., no connection to the RCX must be present and the simulator is used instead. For convenience, there is also a shell-script in the project’s script directory which executes the above.

B.2 DirectMode Shell

The DirectMode Shell is a more advanced application that gives the user the chance to work with the RCX in an interactive fashion. A small language has been devised which allows for addressing all of the RCX’s low-level API functions in a shell-like environment.
The shell has two interfaces; it is available either in a graphical version (depicted in figure B.2) or as a console version. The graphical version has additional features which are not available from the console implementation; they can be accessed over the "Extra" menu of the shell. The user can select (i) to produce a batch file (in which case, the input entered at the shell prompt is written to a specified file) or (ii) to log all of the output and results to a file. Logging (of either sort) can also be stopped again by using the menu.

The shell generally offers an interactive environment for working with the RCX, where commands are entered one by one. However, one can also execute batch files. A batch file contains a list of valid statements which must be separated by newlines or by semicolon, if multiple statements are given on single line. The batch file is processed as if the commands were entered one after another on the shell's input line. By creating batch files and thereafter processing them, it is possible to write small programs that are executed by the shell. For the execution of batch files an extra wait command has been implemented, which can be used to delay the execution of the next command. Thus it is possible, for example, to make a differential drive robot move in a square fashion. The following batch file provokes such a behavior:

```
set motor A state forward; set motor C state forward;
set motor A power 6; set motor C power 6;
wait 3000;
set A reverse;
wait 1000;
set A reverse;
wait 3000;
set A reverse;
wait 1000;
set A reverse;
wait 3000;
set A reverse;
wait 1000;
set A reverse;
wait 3000;
set A reverse;
wait 1000;
set A reverse;
wait 3000;
set motors state float;
```

The general format of a command is `action target [specifier] [value],` abbreviations are possible (set A forward instead of set motor A state forward, for example). The exact grammar of the shell scripting language in EBNF format is listed below (the parser works case-insensitive):

```
Line ::= Command {';'; Command} [';']
Command ::= Read|Set|Activ|Passiv|Lcd|Exit|Help|Batch|Wait
Read ::= read (battery|memory|Sensor|Motor)
Set ::= set MotSpec (Power|State)
Activ ::= activate SenSpec
Passiv ::= passivate SenSpec
Lcd ::= lcd (number|program) Value
Exit ::= exit|quit
Help ::= help
```
B.3 MacroMode Shell

Batch ::= batch <filename>
Sensor ::= SenSpec Mode
Motor ::= MotSpec (power|state)
SenSpec ::= ([sensor] (s1|s2|s3) | sensors)
MotSpec ::= ([motor] (a|b|c) | motors)
Power ::= power <value 0..7>
State ::= [state] (forward|backward|reverse|float|stop)
Wait ::= wait <milliseconds>

Start

The DirectMode Shell can be started with the ApplicationRunner class:

    java ch.ethz.inf.macrobot.ApplicationRunner
        [-simulate] [-graphical] dmshell

By default, the console based shell is started, but the -graphical option selects the graphical UI. The optional -simulate argument starts the DirectMode shell with the selected interface in simulated mode, i.e., no connection to the RCX must be present and the simulator is used instead. For convenience, there is also a shell-script available from the project’s script directory to start the application.

B.3 MacroMode Shell

The MacroMode shell offers the same two front-ends as the DirectMode shell application, i.e., a graphical or console based interface. Figure B.3 depicts a snapshot of the running application using the graphical interface.

![MacroMode Shell Snapshot](image)

Figure B.3: Snapshot of the MacroMode shell

The MacroMode shell is a substitute for the Brain’s Executor component and allows to execute macro mode commands in an interactive way. All macro commands are executed non-blocking. Hence, the user can perform an asynchronous change of the execution level, e.g., start a task while a basic behavior macro is still executing. If, however, a new command is issued before the last one has terminated (without changing the execution level in between), the shell will refuse to execute it.
The exact grammar of the DirectMode shell language is given below:

```
Line ::= Command {';' Command} [';']
Command ::= Begin|Done|Macro|Sensitive|Handler
    |Exit|Help
Begin ::= begin [task]
Done ::= done
Macro ::= forward Distance|backward Distance
    |left Angle|right Angle
Sensitive ::= sensitivity (on Sensor Params|off Sensor)
Handler ::= flush|continue|redo
Exit ::= quit|exit
Help ::= help
Distance ::= <0..65535 cm>
Angle ::= <0..360 degrees>
Sensor ::= s1|s2|s3
Params ::= (a|r) ((u|l)[p]|e) Threshold
Threshold ::= <0..100>
```

Figure B.3 shows an example interaction. If a sensitivity event occurs in sensitive mode, the shell automatically changes into handler mode. To perform a (legal) transition from an execution level with lower priority to one with higher priority (i.e., Basic → Task) use the begin command. done will revert to the previous execution level. Use one of the handler specific commands to terminate handler mode. When entering sensitive mode, the parameters specify mode (absolute or relative) and type (upper, lower, upper pulse, lower pulse or edge) of the sensitivity registration.

Several commands can be issued on one line if they are separated with a semicolon. In contrast to the DirectMode shell a batch command is not supported.

Start

The MacroMode Shell can be started with the ApplicationRunner class:

```
java ch.ethz.inf.macrobot.ApplicationRunner
   [-simulate] [-graphical] mmshell
```

By default, the console based shell is started, but the -graphical option selects the graphical UI. The optional -simulate argument starts the MacroMode shell with the selected interface in simulated mode, i.e., no connection to the RCX must be present and the simulator is used instead. For convenience, there is also a shell-script available from the project’s script directory to start the application.
Appendix C

Implementing Macros

Macros are a central feature of the Macrobot framework. They act as an interfacing layer between hardware (robot-architecture) and software (architecture-independent controller programs). This appendix gives some hints about the implementation of specific macros and sketches the changes that must be performed within the system if any new macro commands are to be added.

C.1 Writing Macros for a new Architecture

A specific robot model cannot be controlled by the Macrobot system, if we do not provide concrete, architecture-aware implementations of the abstract macro commands that it is able to perform (such as forward, backward, turnLeft and turnRight, for example).

C.1.1 Timing

Writing those drivers is not an entirely trivial task. Macro commands typically include a parameter that specifies some spatial displacement, e.g., a distance or an angle. The only means by which we can realize more or less exact positioning of robots according to such commands is by timing. The usage of rotation sensors is not feasible because (i) the design of the Macrobot system does not allow us to employ sensors for any other purpose than generating sensitivity events and (ii) because we do not have any free ports on the RCX to connect any rotation sensors to.

Timing, however, will never be an exact method for measuring displacement of a robot, because it will hardly take acceleration into account. As a general rule of thumb, we can say that the steering of a robot with timing will be more exact if the robot moves slowly, since this reduces the effects of acceleration and inertia. Inertia inevitably becomes an issue when we try to stop a robot immediately. We will never succeed in realizing such behavior if weight and speed of a robot are too high and the force of inertia thus drives our robot further, albeit our halting of all motors at the right time.
C.1.2 Slicing

Apart from the timing issue, we also have to think of the fact that macros must execute in slices, so that their execution can be controlled by the MacroEngine. This may introduce additional complications when implementing a concrete instance of a macro command. Generally, we can say that the controlling of a robot will be more accurate if its macro implementations use small slices. A clever usage of the init and finish methods will help in outsourcing the parts of a macro that are not "sliceable".

C.1.3 Architecture-specific Parameters (Tuning)

The implementation of new macros will always be based on hard-coded architecture-specific parameters. The example-macro listing in section 3.1.4 contained a VELOCITY constant which encoded the distance that the target model could cover in a certain amount of time. Naturally, most macro-implementations will be based on similar constants.

The challenge for a macro programmer is to determine those constants. However, this can not be done before an actual physical robot instance of the specific architecture is available, equipped with the exact load that it will be bearing when going into action. The latter is very important since changes of load, size, weight or other properties will affect the robot’s speed and behavior when performing a specific maneuver.

How does one fine-tune a macro implementation, i.e., determine the architecture-specific parameters? For any macro implementations that work on the basis of timing (and most of them will, due to the reasons specified above), the robot’s performance/time ratio must be measured when executing the macro-task over a longer period of time. We can, for example, measure the distance that a robot covers in 4 seconds several times and then compute the mean velocity from those values, that is, the distance that it covers in one second.

A small Case Study

Let us illustrate the process of fine-tuning a macro implementation with a small case study. We will determine the velocity of a differential-drive-robot driving forward.

Before we measure the robot’s speed, we should think of how the macro will be implemented. Our robot possesses two motors, connected to port A and C of the RCX, respectively. Driving forward is simple: powering both motors identically will do the job. So we write a small LeJOS program which actually makes the robot drive forward for a specific period of time:

```java
import josx.platform.rcx.*;

public class TestTurnLeft {

    public static void main(String[] args) {
        // init params
        int POWER = selectValue(3, 1, 7);
        int TIME = selectValue(2000, 9000, 5000);
```
// make the robot spin counterclockwise
Motor.A.setPower(POWER);
Motor.A.backward();
Motor.C.setPower(POWER);
Motor.C.forward();
try { Thread.currentThread().sleep(TIME); } 
catch (InterruptedException e) {} 

// stop the machine
Motor.A.stop();
Motor.C.stop();

public static int selectValue(int val, int inc, int max) {
    int RUN = 0x01;
    int PRGM = 0x04;
    int mask = Button.readButtons();

    show(val);

    // PRGM changes value, RUN selects
    while ((mask & RUN) != RUN) {
        if ((mask & PRGM) == PRGM) {
            val = (val+inc) % (max+inc);
            show(val);
        }
        try { Thread.currentThread().sleep(200); } 
catch (InterruptedException e) {} 
    }

    return val;
}

public static void show(int val) {
    MinLCD.setNumber(0x301f, val, 0x3002);
    MinLCD.refresh();
}

We’ve written the program in a general way, so that we can parametrize execution with respect to the motor power we wish to use and to the time in milliseconds that the program should run. By default, the motors will receive power 3 and run for 2 seconds. When the program is run, the user can first select the power for the motors: pressing PRGM will cycle through all possible values (0..7), pressing RUN selects the value to use. Subsequently, the time to run must be chosen before the algorithm executes with the given parametrization.

After compiling the program with lejosc and uploading it with lejosrun we can start measuring the performance of our model.

On a desk or on the floor we have marked a straight line and indicated the starting position of the robot. After setting the robot on its start mark, pointing
Implementing Macros

into the direction of the line, we select power and time to run (press RUN twice
to use the default values). Then we let the machine run forward until it stops
and measure the distance that it has travelled. We repeat this procedure several
times to get more data, maybe we also run tests with different parameters.

Once we’re satisfied with the data that we’ve collected, we divide the mean
distance travelled by 1000. The resulting value is the velocity of the robot in
cm/sec.

Next, we write our actual macro, which in the end, may look just like the
example macro listed in section 3.1.4. Before we employ the macro, we should
test and control its accuracy with a program like the following one:

```java
import josx.platform.rcx.*;

public class ForwardTest {

    public static void main(String[] args) {
        int DISTANCE = getValue(50, 5, 200);
        ForwardMacro fw = new ForwardMacro();
        fw.execute(100, DISTANCE);
    }

    public static int selectValue(int val, int inc, int max) {
        // ... listing omitted, identical to above
    }
}
```

After the selection of an arbitrary distance, the program will execute our macro
implementation with that distance as parameter. Short distances will most
likely be off, because the timing method doesn’t usually take acceleration into
account; the actually travelled distance will thus be smaller than calculated.
There’s not much we can do about this, unless we measure acceleration as well
and use this knowledge when implementing a macro. If the macro is off only a
bit, we might fine-tune it further by tweaking the constant parameter value “by
hand”. Often though, other factors will influence the accuracy of a macro, such
as the quality of the ground the robot moves on.

In the end, ”the art of macro tuning” mostly consists of finding a set of
parameters that is optimized for best accuracy in the average case. After all,
the scenarios that we will be investigating with the Macrobot framework will
not require a very precise positioning of robots.

Other macros, such as turn left, are implemented similar to the case we just
discussed, only that one uses a goniometer rather than a ruler to measure the
distance travelled.

Overcoming the Limitations of Timing

As we’ve stated before, timing is a rather inaccurate method for calibrating
macros, especially for fast robots. If a specific macro implementation happens
to be totally off, we can possibly improve exactness by altering the robot’s
architecture, e.g., by gearing down to consume speed in exchange for torque.
Be careful though, gearing down too high may damage motors, in case of stalling.
C.1.4 Configuring the Spine

All the macro implementations of a specific architecture should be logically grouped in an own package, `spine.architecture.arch-name` is the convention. Across architecture packages, macro implementations should always be given the same name if they perform the same task.

Before uploading the Spine to the RCX of a new robot model, we must configure the `MacroFactory`, so that it produces the correct macro instances. This is done by changing the import statement at the beginning of the class file to

```java
import ch.ethz.inf.macrobot.spine.architecture.arch-name.*;
```

where `arch-name` should be replaced with the package name of the robot-specific architecture. Once that we’ve recompiled the `MacroFactory` class, we can upload the Spine subsystem. Due to the new dependency that we just introduced, the linked binary will automatically contain the correct macro classes.

C.2 Adding a new Macro Command

Adding new macro commands (e.g., `grab` or `drop` for picking something up and dropping it again) implies the following changes within the `Macrobot` framework (class names are given relative to `ch.eth.inf.macrobot`):

1. Think of the syntax of the command. It should not have more than two arguments of byte size (or a single argument of short size).

2. Add the new command to the `brain.macromode.MacroCommands` interface (i.e., add a new method).

3. Add a facade method for the new command to the `brain.macromode.MacroProgram` class.

4. Add an implementation of the new command method to the `brain.macromode.MacroCommandsProxy` class and encode the parameters properly (have a look at the other command implementations to get an idea about how this should be done). Use the conventional encoding for parameters! Choose an unique ID for the new command and add this information to the globally shared `MMConstants` interface.

5. Extend the `Nil` device implementation of the Brain package so that it immediately returns a correct ack packet if it receives an execution command for the new macro.

6. Write a macro implementation that extends the abstract `spine.Macro` class and put it into the `spine.architecture.arch-name` package.

7. Edit the `getMacro` method of the `spine.MacroFactory` class, so that it produces an instance of the new macro implementation if it is given the right ID.

8. Recompile the framework and upload the Spine to the RCX. With some luck there will be enough free memory to hold the additional macro implementation.
9. Be aware of the fact that macro programs which were compiled with the old version of the framework will not run on the new version, unless they are also recompiled. This is a natural consequence of the fact that internal interface definitions have changed.
Appendix D

Prototype Robot Model

The following pages contain building instructions for the construction of a Carriage-Bot.

This prototype device is equipped with a rack that can hold both an RCX and a complete iPAQ together with sleeve and a wireless LAN card. The display of the iPAQ can be accessed by swinging up the RCX. The construction has been deployed for mounting the IR tower as well (just in front of the RCX’s IR LED). If using RCX-IR (i.e., a direct IR communication between the iPAQ and the RCX, bypassing the tower) one can simply mount a piece of a CD in place of the tower, this has been reported to reflect the infrared signals rather well.

The Carriage architecture is symmetrical and possesses a caster wheel and a bumper sensor on each side. The drive is a differential one and lets the robot turn in place. On the bottom side we find a light sensor monitoring the ground. Despite its somewhat clumsy design, the robot is quite well maneuverable, at least on a smooth surface. It achieves high speed, but acceleration is slow, due to its weight.

Note The instruction pictures were generated with MLCAD [23], a CAD application, exclusively designed for building LEGO models. However, flexible parts cannot be modelled in MLCAD. Hence the pictures are missing some elements, namely rubber bands that keep the bumper sensors in a normally closed state and the sensor cables. For the experienced builder it should be quite obvious where those missing elements must be added.
Prototype Robot Model
Bibliography

Books

Articles

Internet
BIBLIOGRAPHY

[13] RCX Port API:
   http://www.slewis.com/rcxport/

[14] RCX Java API:
   http://www.escape.com/~dario/java/rcx/

[15] TinyVM project homepage:
   http://tinyvm.sourceforge.net

[16] RCX-IR project homepage:
   http://rcxir.sourceforge.net

[17] RCX internals, reverse engineering data:
   http://graphics.stanford.edu/~kekoa/rcx/

[18] Aspect Oriented Software Development:
   http://aosd.net

[19] Programmable Software Extensions (PROSE):
   http://www.prose.ethz.ch

[20] Jini(tm) Network Technology:
   http://www.sun.com/software/jini/

[21] Mindstorms resources:
   http://www.crynwr.com/lego-robotics/

[22] GNU ToolChain Project:
   http://www.handhelds.org/projects/toolchain/

[23] Mike’s LEGO CAD (MLCAD):
   http://mlcad.ldraw.org