ISSN 0280-5316
ISRN LUTFD2/TFRT--5740--SE

Teleoperation and Autonomous Navigation of a Mobile Robot Using Wireless Vision Feedback

Jens Graf

Department of Automatic Control
Lund Institute of Technology
March 2005

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
March 2005

Document Number
ISRN LUTFD2/TFRT--5740--SE

Author(s)
Jens Graf

Supervisor
Karl-Erik Årzén and Anders Blomdell at LTH in Lund.

Sponsoring organization

Title and subtitle
Teleoperation and Autonomous Navigation of a Mobile Robot Using Wireless Vision Feedback. (Färrstyrning och autonom navigering av en mobil robot med trådlös kameraåterkoppling)

Abstract
At the department of Automatic Control at Lund Institute of Technology a mobile robot has to be assembled. The capabilities of the vehicle comprises a possibility to control it via internet with the help of a Java-applet as well as a function that realizes an autonomous navigation through a predefined environment.
The first part of this work deals with the implementation of the remote control function. This involes setting up the hardware components and the realisation of different interfaces between these components.
Another mode in which the robot can be driven ensures autonomous navigation in a known environment. Since the environment the robot is moving in is a corridor, it is a crucial part to guarantee the straight driving of the robot. Also, a simple map and the recognition of landmarks are necessary to orientate in the robots surroundings. In this context, pattern recognition is done by calculating the correlation coefficient.

Keywords
Mobile robotics, Etrax, Atmega, navigation, pattern recognition, correlation coefficient

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
117

Recipient's notes

Security classification

Contents

1. Introduction **6**

2. Problem Description **9**

3. Physical Setup **13**

3.1. The Etrax Computer 13
3.2. The Network Camera 14
3.3. The Robot Platform 14
3.4. The Atmel Controllers 15
3.5. Wireless LAN 17

4. Interfaces and Programming Techniques **18**

4.1. UDP Socket Programming 18
4.1.1. C-Server 19
4.1.2. Java Client 21
4.2. RS232 Serial Port 22
4.2.1. Accessing the Serial Sort on the Etrax Computer 23
4.2.2. Accessing the Serial Port on the Atmel Controller 25
4.3. Two Wire Serial Interface - TWI 27
4.3.1. Starting and Stopping a Data Transfer 28
4.3.2. Address and Data Packet Format 29
4.3.3. Transmission Modes 31
4.3.4. Master Transmitter and Slave Receiver 31
4.3.5. Master Receives and Slave Transmits 39
4.4. Multithreaded Programming 42
4.4.1. Threads Implemented in C 42
4.4.2. Threads Implemented in Java 43

5. System Assembly **45**
5.1. Communication Protocol between the Components 45
5.2. Channel and Command Selection 47
5.3. Information Exchange between the Java-Applet and the Etrax Computer 48
5.3.1. Sending data from the Java-applet to the Etrax-computer 48
5.3.2. Sending Data from The Etrax Computer to the Java-Applet 51
5.4. Information Exchange between the Etrax Computer and the Atmega16 Controller 51
5.4.1. Sending Data from the Etrax to the Atmega16 51
5.4.2. Sending Data from the Atmega16 to the Etrax 51
5.5. Information Exchange between the Atmega16 and the Atmega8 53
5.5.1. Sending Data from the Atmega16 to the Atmega8 53
5.5.2. Sending Data from the Atmega8 to the Atmega16 54
5.6. Pulse Width Modulation to Drive the Motors 54

6. Remote Control - Teleoperation **57**
6.1. Generating Motor Commands from the Java Control Panel 57
6.2. Results 61

7. Autonomous Navigation **62**
7.1. The Robots Abilities and the Environment it is Moving in 63
7.2. Correlation Coefficient - Mathematical Basics 64
7.3. Motor Controllers - Straight Movement 66
7.3.1. Recursive Control Algorithm 67
7.3.2. The Motor Control System 68
7.3.3. Implementation on the Atmega8 68
7.3.4. Track Correction at Regular Intervals Using Pattern Recognition . 72
7.3.5. Results 74
7.4. Navigation in the Corridor 75
7.4.1. Recognition of a Lamp 75
7.4.2. Windowing the Image and Finding the Pattern 76
7.4.3. Simple Navigation - Counting the Lamps 79

8. Conclusion and Outlook **82**

9. Acknowledgement **84**

A. The Matlab Function "corrco" 86

B. Source Code of the Atmega8 Program 88

C. Source Code of the Atmega16 Program 97

D. Source Code of the Etrax Program 101

E. Source Code of the Matlab Program 107

F. Source Code of the Java-applet 112

1. Introduction

The term "robot" was coined in a play called "RUR," by the Czech writer K. Capek. In this play, he named mechanical human beings "robots." These "robots" stood at workbenches instead of human beings. He derived the word robot from the Slav word "robota," which means work. Nowadays there are mainly two domains in robotics: industrial and mobile. Industrial robotics works with stationary robots, and its intention is to automate industrial manufacturing processes. On the other hand, the mobile robotics intends to provide an assistant machine to human beings. Robots should facilitate our daily life. There are a vast amount of application areas ranging from personal assistant robots to mobile manufacturing robots. To develop these types of machines it is necessary to have capable algorithms that realize autonomous functions. The machine must be able to make its own decisions, taking into account not violating a human being. All of this necessitate intelligent algorithms. Even if we have good approaches to realize intelligent behavior, we are far away from having machines which are able to make their own decisions. Nevertheless, contemporary robots have absolutely useful abilities. For example contemporary industrial products like autonomous vacuum cleaning robots or mobile robots that are used for transportation in storage buildings already exist. Since sufficient intelligent algorithms are not available most mobile robots need a large amount of electronics and sensors. Besides it is necessary to achieve a proper functionality in all conceivable situations of operation. Another important issue is that it is not allowed to harm human beings. Since all these issues have to be considered, mobile robots become very expensive and therefore it is not economical to produce them.

In the 1980s, the Department of Automatic Control at Lund University of Technology bought a mobile robot. It was equipped with bumper sensors as well as ultra sonic sensors. It was primarily built for educational reasons. In 2004 it was decided to redesign this robot. The majority of the old components are no longer used. However, the chassis with the bumper sensors and the motors are used. The core unit of the new robot consists of an Etrax 100LX computer [1], which runs an embedded Linux system. Two Atmel microcontroller boards are used as well. One microcontroller board is used to

read sensory information and another one controls the motors. Additionally, a camera should be mounted on the top of the robot. The camera is connected to a wireless LAN which is accessible from the internet.
The first task is to implement a remote control function. One should be able to log in on the robot. A Java-applet provides a user interface to control the robot and move it. Commands for moving will be sent from the user host to the embedded Linux computer on the robot. Status information, such as speed, will be sent from the robot to the user host. A second task will be the implementation of autonomous functions with the help of image processing. The robot has to be able to move on its own. Therefore, it uses a given map of the environment in which it is moving. With object recognition, the robot will recognize where it is located. Using this, it can check its actual position against a target position predefined by the user and find its own way through the environment. An important condition is that the environment is not too complicated.

Outline of the Report

Chapter 2 introduces the basic problems of this work. The components of the robot are briefly described and a description is given on how the individual parts work together. Chapter 3 deals with the robot's individual components in detail. The functions of the single parts are described as well as their link to peripheral components such as sensors and motors. Most components of the robot exchange information through different interfaces. These interfaces and how to access them as well as programming techniques are described in Chapter 4. In the following Chapter 5, is explained how the assembly of the individual parts is done. The Etrax computer, an embedded Linux system, will be a crucial component. All communication between a user and the robot occurs through the Etrax computer.
Two Atmel controller boards are connected to the Etrax computer. They are responsible for communication and control of the sensors and motors. The Etrax computer, as well as the network camera, are both involved in a wireless LAN network. A host computer, which is also connected to the wireless LAN, provides enough computing power to perform speed demanding calculations. Chapter 6 describes how the robot can be conrolled by a user over the internet. A Java-applet provides an interface that makes is possible to control the robot easily with the computer mouse.
Chapter 7 describes the problems concerning autonomous navigation and provides basic approaches to achieve simple navigation in a known environment. It is pointed out how important it is to ensure to drive straight. In connection with that, a motor control

system consisting of PI-controllers is developed. This control system is supported by pattern recognition. It is shown how navigation can be done by the implementation of a simple map. To orientate, the robot uses the known position of landmarks that are recognized by calculating of the correlation coefficient.
A conclusion and a summary of the project is provided in Chapter 8. Also, an outlook for possible future work is given.

2. Problem Description

At the department of automatic control at the Lund University of Technology exists an old mobile robot called RB5X []. Since the robot was not in use for the last few years, it was disassembled. It is necessary to assemble all these parts to get the robot to work again. One of the robot's core units consists of a 100Mhz Etrax RISC computer which is responsible for the communication between the hardware components. The Etrax computer includes a network socket to connect it to conventional computer networks. In addition to that, a Linux system is installed on it. The network socket is used to communicate with a Java-applet running on a host computer connected to the internet. The applet makes it possible to log on to the robot from all over the world. The RS232 port on the Etrax board is used to communicate with the robot's hardware, e.g. two Atmel controller boards. Since the local computer power is restricted, more power demanding operations have to be done by an external host. However, this host is not the computer the Java applet runs on. In fact it is a computer connected to the robot's LAN.
Also connected to the Etrax computer are two Atmel controller boards. One of the Atmel controller boards is responsible for reading sensory data from 16 bumper-switches. These bumper sensors are used for security reasons. When the robot is hitting an obstacle the bumper-sensors should indicate the presence of an obstacle and appropriate action to take. The second Atmel controller-board provides an interface to control the motors. It will be connected to the Etrax-computer and receive motor-commands via RS232 serial port. It is necessary to define these commands and the protocol for the communication. It is of utmost importance that in every data packet, a command, the address of the receiving hardware and a value are sent. Since all data packets will be delivered via the Etrax computer, a program has to be launched to distribute the data to the right device. The communication and the drive control have to be implemented.
A network camera is also involved in the wireless-LAN. This camera will be located on the robot's head to observe the environment. The network camera has a TCP/IP socket to connect it to a network. The address of the image has to be linked to a webpage located on the Etrax web server to enable a user to see the camera image when he visits

the website.
The first task is to get all the individual parts work and implement several communication interfaces between the hardware components reaching from RS232 connection over TWI to TCP/UDP socket connections. A serial communication interface between the Etrax computer and one Atmel controller board has to be realized. The two Atmel boards communicate over a TWI bus which has to be programmed. The Etrax and the host computer transfer data via the UDP internet protocol. Since the Etrax computer is an integral part of the communication, it has to distribute incoming data to the appropriate hardware. The work also involves the control of the motors. When all the hardware is working and a smooth communication between the components is bridged, the main objective is to realize a remote-control function. The robot should be controlled via internet. Therefore, a webpage hosted on the Etrax computer has to be written. This webpage is downloadable from the Etrax web server and the camera image from the robot camera can be seen on this webpage. After invoking the website an included Java-applet will be started on the website. This applet provides a control interface to send commands to the robot. The commands comprise instructions to control the motors of the robot as well as request commands to get status information. Thus, the user can steer the robot from any computer connected to the internet. Status information, for instance, the robot-speed will be sent back to the Java-applet and will be displayed in the user's web browser. Because of the robot's network camera the user is able to see the environment of the robot on his web browser. The right to access the robot is a crucial thing and has to be treated in this work. The mobile robot with its components can be seen in Figure 2.1. A schema of the whole robot system with its mobile and stationary parts is given in Figure 2.2.

A second task is to realize an autonomous mode. In this context, it is necessary to ensure that the robot is able to drive straight. Providing the motors with equal PWM signals leads not to straight driving, rather driving along a curve since the motors are not completely equal. Therefore, different controllers have to be developed. With the help of image processing the robot will be able to find landmarks in the environment the robot is operating in. This will be done with pattern recognition. Based on a simple map the robot knows where it is located and thus can find its way to a predefined destination, for instance, a special room. Since the computing power of the devices located on the robot is too low, the image processing will be done on an external host computer.

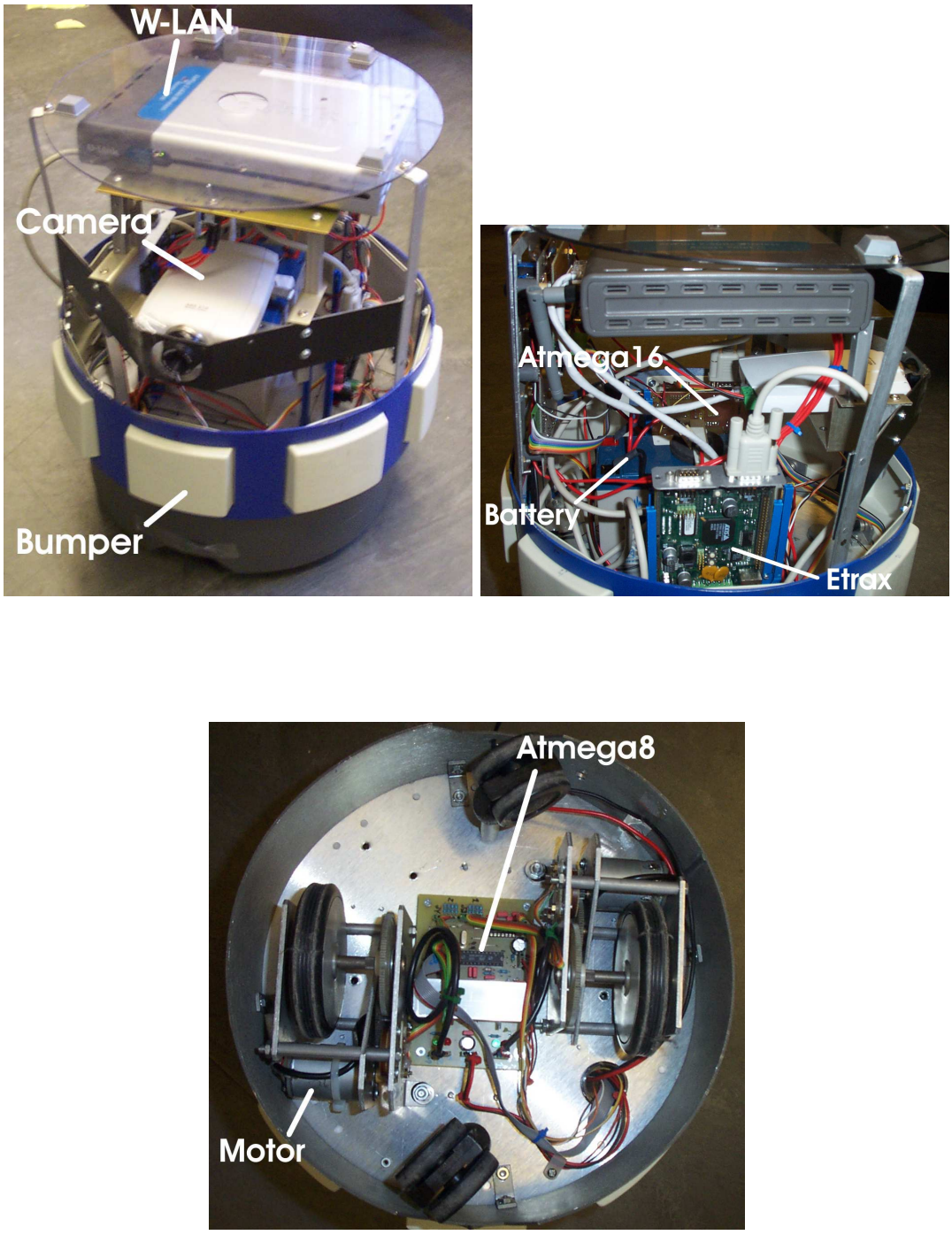

Figure 2.1.: Different views of the mobile robot.

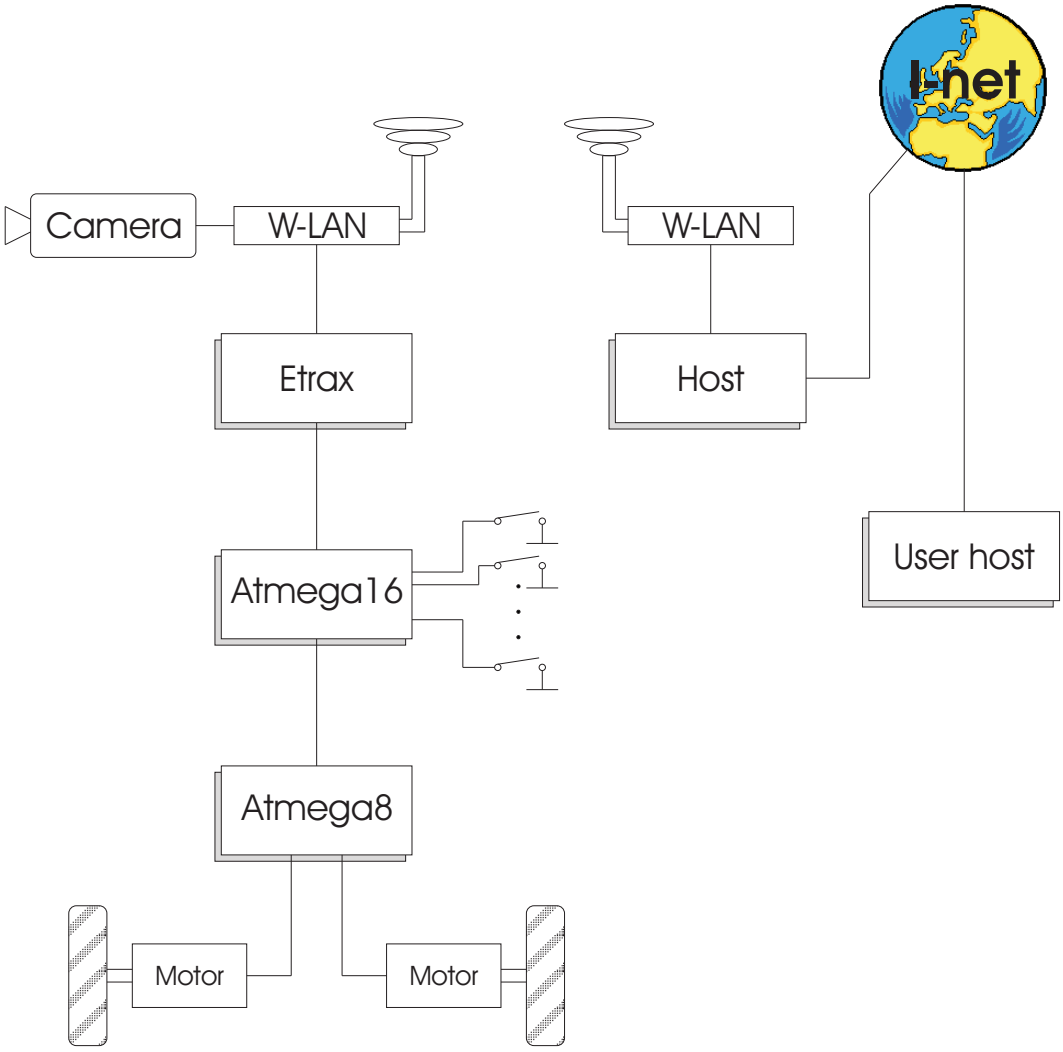

Figure 2.2.: Schema of the robot hardware.

3. Physical Setup

3.1. The Etrax Computer

The Etrax computer, developed by the company Axis Communications AB, is specially designed for embedded Linux development. It has integrated Ethernet and thus it is possible to connect easily to a wireless LAN. The 100 MIPS 32 bit RISC CPU provides enough speed to fulfill the required tasks. Two synchronous and four asynchronous serial ports are available on the chip. In this work only two of the serial ports are used. One of the Atmel controller boards is connected to one serial port. The Etrax chip is mounted on a special developer board. Two serial ports are accessible over 9 pin SUB D outlets. As mentioned before, one serial port is used to connect the Etrax computer to one controller board. The other serial port is needed to connect to the host computer. Through this port it is possible to transfer and start programs on the Etrax computer. The Etrax computer provides 8MB of RAM and 4MB of flash memory whereby 2 to 2.5 MB are available for the applications. To write C-programs for the Etrax computer, a special compiler is necessary. The program development is performed on a personal computer. After writing the source code, it is necessary to use a special compiler called Cris compiler to compile the source code. This compiled program is executable on the Etrax system. To execute the program, it has to be transferred via FTP to the flash memory on the Etrax computer. Now, the file must be changed to executable mode and the program will be able to run with the Linux common command to execute a file. On the Etrax computer, a special version of a Linux kernel is running. This Linux is available on the producer's homepage [3]. It has to be compiled and flashed to the board. The boa Ethernet server is included in the Linux system, hence, a user can easily put HTML files via FTP on the Etrax computer. These files can be reached over the board's internet address. On the server, a HTML file will be available which displays the camera's image and a Java-applet to steer the robot. Picture 3.1 illustrates the single components of the Etrax computer which are used in this project.

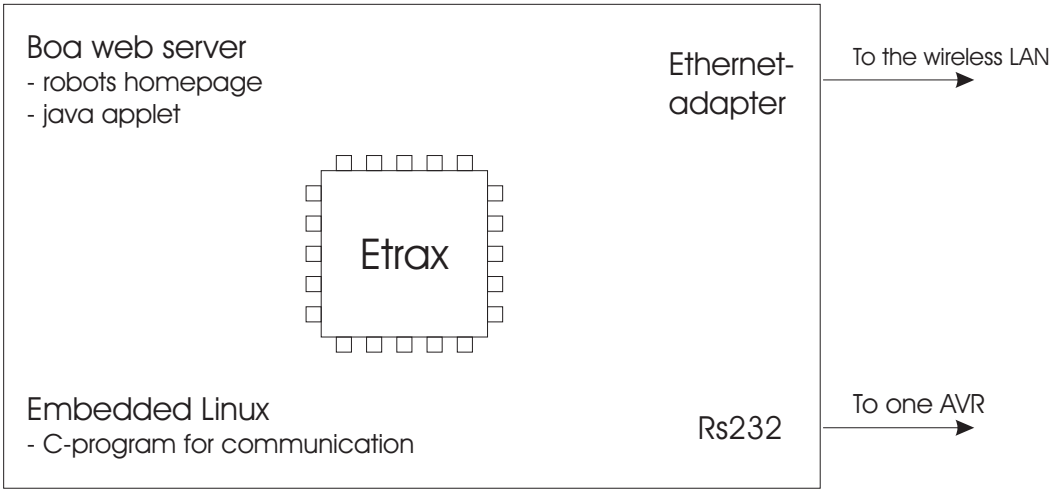

Figure 3.1.: Important components of the Etrax computer.

3.2. The Network Camera

On the robot, an Axis 210 network camera is used because it can be directly plugged on to a network. It has a built-in web server on which single images or video streams are accessible. To use the camera's images, a link has to be set to the video file located on the camera's web server. The camera is mounted on the top of the robot. The camera is very important for functions such as the implemented remote control of the robot and autonomous operations which will be described later. For the remote control function, a user invokes a HTML page located on the Etrax computer. This page includes a reference to the video stream on the camera's web server. Thus, the user can view the images taken by the camera on his web browser. The camera's images are also very important for the autonomous operation of the robot to obtain information from the environment. The camera's images can be used to find special marks in the surroundings. These marks can be used as way points to realize automatic orientation of the robot.

3.3. The Robot Platform

The platform for this work is delivered by a robot called RB5X. It is an old machine manufactured by General Robotics [2] in 1983. This robot was produced for experimental and educational purposes. The original version of the robot is equipped with an INS8073 microprocessor from National Semiconductor with 4MHz system frequency. It can be programmed in Basic language and there is a number of additional teaching softwares

available which makes it possible to teach the robot basic behaviors. It has Polaroid Rangefinder sonar sensors to detect obstacles in its way. The detection range of these sensors can be set from 10 inches up to 35 feet. Eight tactile sensors are mounted around the robot. They are able to sense a contact when the robot bumps against an object. Furthermore, a manipulator with four links and one gripper is available. Therefore, it is possible to grab objects in the robot's surroundings. An important aspect of the robot is that it is modularly designed. Hence, one can have easy access to the single components of the robot. For this project, the basic framework of the old robot is used. Only the Motors, the chassis and the bumper switches are used. The manipulator is used as well but its controlling and activation is done by a parallel project. Two Atmel AVRs are added to control the motors and to read the bumper switches. An Etrax computer is responsible for the general communication between the components. In addition, the Etrax computer is connected to a wireless LAN which involves a host computer to carry out demanding computational calculations. A network camera is mounted on the top of the robot to deliver live video images.

3.4. The Atmel Controllers

The project includes two Atmel AVRs with 8 bit RISC architecture. The Atmega8 has 8 Kbytes flash and the Atmega16 has 16 Kbytes flash memory which is sufficient to store the programs. The available peripheral components comprise among others, a programmable serial port, TWI interface and PWM channels. The Atmega8 is responsible for receiving motor commands from the Etrax computer. These commands are interpreted and converted to PWM signals which are sent to the motors. The controller also reads pulses from encoders which are mounted on the wheels. These encoders provide information of the robot's velocity. A simplified circuit diagram of the Atmega8 is depicted in Figure 3.2. Only the important pins are shown to keep the clarity. The scheme shows a MAX202 which is responsible for driving the serial port. Mainly it converts the 0V to 5V amplitude to a -12V to 12V amplitude which is common for serial RS232 ports. The port pin Pd0 is connected to the receiver line and the pin Pd1 is connected to the sender line respectively. On the right side of Figure 3.2 the motor drivers can be seen. To each driver lead five connections. To the first driver unit lead the pins Pc2, Pc0, Pd6, Pb1 and Pb3. Pc2 determines the left motor's direction. The current sense output is connected to pin Pc0. The motor driver delivers at this output a current of 377A per ampere motor current. To read a voltage with the analogue-digital converter

at Port C, the current sense output is connected to a resistor. With the break signal on Pd6 is it possible to short the H-bridge of the motor driver. The Atmega8 is equipped with PWM signal generators. Pin Pb1 is one PWM output and it is connected to the left motor driver. If the motor driver becomes too hot, the thermal flag on Pb3 is set. The flag is set at a temperature of 145C but the circuit will not be shut until the temperature reaches 170C. The pins Pc3, Pc1, Pd7, Pb2 and Pb4 are connected to the second motor driver. The functions and connections to the motor driver are the same as for the previous motor. Two encoders provide a signal proportional to the wheel speed. Each encoder delivers a phase shifted signal. The signals of encoder "left" are connected to the pins Pd2 and Pd4 and the signals of encoder "right" are connected to the pins Pd3 and Pd5 respectively. The Atmega8 provides also support for connecting to a TWI serial bus. The pins Pc4 and Pc5 are lead through to connect to the TWI bus; external circuitry is not shown in the simplified circuit diagram. Pc6 and Pb5 lead to the programming connector.

The Atmega16 provides an interface to the bumper switches. The switches are read and the data is delivered to the Etrax computer for further analysis. Another task of the Atmega16 is to transfer information from the Atmega8 to the Etrax computer and vice versa. The two Atmel AVRs exchange data through a TWI interface and the Atmega16 is connected via RS232 to the Etrax computer.

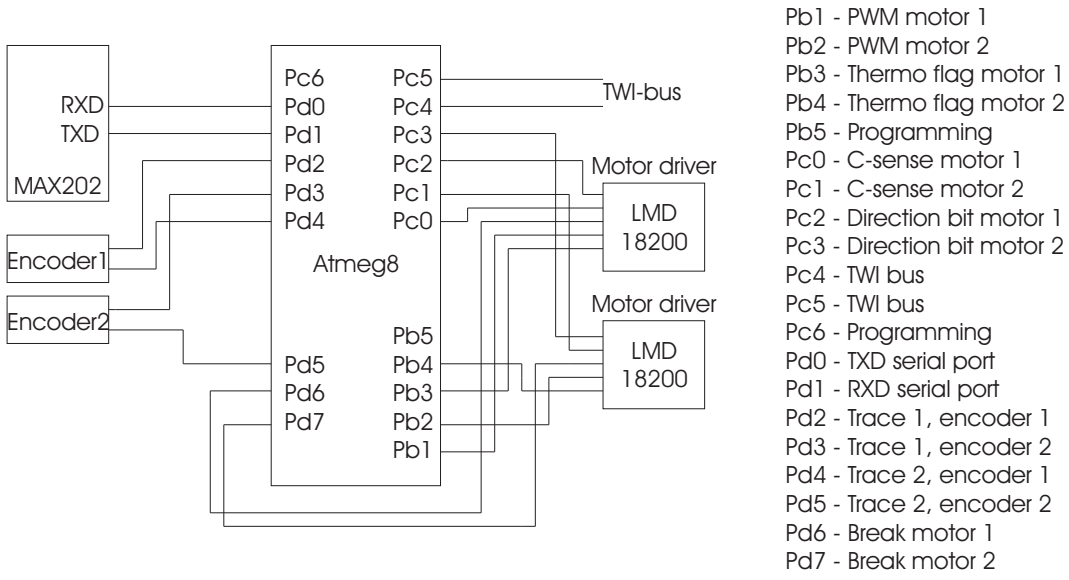

Figure 3.2.: Peripheral components connected to the Atmega8 controller.

3.5. Wireless LAN

Since the robot should be accessible via the internet, it is necessary to have a network connection to the robot. Therefore, wireless LAN adapters are used to provide a wireless connection from the robot to a host computer which routes requests further to the internet. So it is feasible to log on the robot via wireless LAN from the internet. Also, the network camera is connected to the wireless LAN adapter on the robot. The wireless LAN adapters provide the wireless connection to realize the remote control with the Java-applet. Adapters from type DWL-1000AP+ are used since they provide diverse possibilities to make a connection secure. For this application it is necessary to guarantee that only authorized users have access to the robot. Therefore, it should be impossible for others to access the wireless network.
The wireless LAN adapters have two options to ensure security. On one hand a MAC address selection is provided. So just authorized devices have access to the adapters. Every adapter in the LAN has a list with authorized devices. So every single device knows if a request of a device comes from an authorized device or not. Devices not belonging to the network are blocked. On the other hand the adapters support a standard called 802.1X which is a standard for encrypted communication. An encryption key up to 256 bits is possible. This is an additional possibility to make the wireless network save. Since both methods provide security enough to guarantee that only authorized users have access to the robot, they are used in this work.

4. Interfaces and Programming Techniques

4.1. UDP Socket Programming

There are mainly two protocols existing, which make it possible to send data across the internet between two different computers or devices. TCP (Transmission Control Protocol) is a connection based protocol, which means that a connection is opened and established between two points in the net. Then, data can be sent and received with the reliability that all data is sent and received with the right timing. To make the data transfer reliable, the protocol includes an acknowledgement of the data which leads to higher network traffic. In case of transmitting problems, errors indicate the loss of information. When all data is sent and the communication is ended, the connection has to be closed.
In contrast to TCP, UDP is not connection based. The communication is not guaranteed in regard to time or the delivery of data. Also, there is no guarantee that data packets are delivered in the same order they were sent. Therefore, the UDP protocol belongs to the unreliable internet protocols.
UDP transfers data by sending independent packets called datagrams. This is similar to sending a normal letter through the postal service. Here, it is not guaranteed if a letter is delivered or not. Additionally, every letter is independent from each other.
Both protocols have different drawbacks. TCP loads the amount of traffic on the net. However, it guarantees a delivery of sent data in the right order.
The robot will be controlled from a Java-applet. This applet needs to exchange data with the Etrax computer over a network protocol. Typical data, such as commands which set the robot's speed, are sent to the Etrax computer. But, this is not crucial information. Since the command will be sent continuously and the next command reaches the receiver in a few milliseconds, no role is played if a command is either received or not. Because it does not matter if every single command is received, taking into account that it is necessary to keep the net loading as small as possible, UDP is used for network communication between the Java-applet and the Etrax computer. In the case that

communication fails, it has to be recognized by the receiver and be taken care of. This could be done with a kind of watch dog timer implemented in the receiver's program. If a loss of communication is not recognized, the robot will probably run into obstacles and this could result in damage. A UDP connection is usually established using the client server model. One process, the client, connects to the other, the server. So, the server has to set up a connection and the client just links to the server in order to request information. This is similar to a telephone call where one conversational partner dials a number and waits for the other to pick up. But, the difference from a telephone call is that the server does not need to know the client's address. The server does not even need to know if the client exists. A connection between a server and a client is realized through sockets. A socket is a one end-point of a two way communication between processes.
Java and C implement sockets with special socket classes. On the Etrax computer, a server programmed in C waits for the Java client to request information. When the client sets up a UDP socket connection, it can create a datagram packet. This packet will be sent over the socket connection to the server. Since it is a two-way connection, the server can reply with another datagram packet. The following sections describe how to realize a server in C programming language and a client in Java.

4.1.1. C-Server

In order to initiate a connection, a socket has to be created:

```
sd=socket(AF_INET, SOCK_DGRAM, 0);
```

The command "socket" returns an integer value which is assigned to the descriptor of the socket. This descriptor is used to access the socket. The function "socket" expects three parameters. The first parameter AF_INET is the domain parameter and specifies a communication domain. It selects the protocol family. The protocol is defined by parameter three. Since there is just one protocol available in the protocol family this parameter is set to zero. The second parameter specifies the socket type. With SOCK_DGRAM it is determined that datagrams are supported. To assign a port to the server socket the "bind" command is used:

```
servAddr.sin_family = AF_INET;
servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
servAddr.sin_port = htons(LOCAL_SERVER_PORT);
```

```
rc = bind (sd, (struct sockaddr *)
&servAddr,sizeof(servAddr));
```

The first argument passed to the "bind" function, is the socket descriptor of the server socket. Also, "bind" needs to get an address of a structure of the type "socketaddr_in". In this structure the protocol family, the server address and the port has to be defined. AF_INET is the same parameter that was passed to the "socket" command. The function "htonl", defined in the header file "<netinet/in.h>", converts the port number to a capable sequence which is different on different machines. On some machines this function is defined as "Null" macros and on other architectures it has a function. LOCAL_SERVER_PORT is user defined at the begin of the program and assigns the port for the socket. The last parameter is the length of the structure. Now, the server socket is ready to receive datagram packets from the client. Since a "recvfrom" command, which receives packets from the socket, waits endless time, the receive function is realized in a separate thread. So other running processes are not blocked if no data is received.

```
void *receiveUDP(void *arg) {
  cliLen = sizeof(cliAddr);
   while (1) {
     memset(&msg_r, 0, sizeof(msg_r));
     n = recvfrom(sd, msg_r, MESSAGE_LENGTH, 0, ...
              ... (struct sockaddr *) &cliAddr, &cliLen);
     if (strcmp(msg_r,"<<init>>")==0)
       { continue; }
     else {
       printf("receivedUDP: %s %d\n",msg_r,strlen(msg_r));
       RS232_send(msg_r);//send it further via RS232 !!!
     }
   }
  return NULL;
}
```

The array "msg_r" is an array of type char to store the received data. The "memset" command sets all elements of the array to zero. This is necessary since a zero indicates the end of a string. The "recvfrom" command expects six parameters. The socket descriptor is necessary as well as an array to store the received data. MESSAGE_LENGTH is the

length of the array and tells the "recvfrom" command how many bytes will be received. With the fourth parameter it is possible to pass the function a flag but this is not done here so it is passed zero. The fifth parameter specifies the address of a struct in which the source address of the received massage will be stored. Basically, a UDP socket connection is a two way connection, that is the socket and the client can receive as well as send datagram packets. But it must be taken into account that the first message has to be send by the client and received by the server. After the reception of that message, a connection is established. The first message sent by the client contains the string "<<init>>". It is not necessary to proceed this string cause it has no meaning. All packets received after the first initial packet will be send further to the Atmel board via RS232. To send data over the UDP socket connection, a function is provided as follows:

```
void sendUDP(char *data) {
  n = sendto(sd, data, MESSAGE_LENGTH, 0, ...
          ... (struct sockaddr *) &cliAddr, cliLen);
}
```

The second argument passed to the function "sendto" is a pointer to an array of chars which have to be sent. The rest of the parameters are the same as it was at the "recvfrom" function. It needs the socket descriptor, the length of the data array, flags as well as the address of a "socketaddr" structure and its length. The function returns the number of sent bytes after it completed the sending without an error. It returns -1 to indicate an error.

4.1.2. Java Client

The client is a part of the Java-applet that the user can download from the Etrax web server.
First, the UDP client has to be initialized. Therefore, a socket of type "DatagramSocket" is created. It is not necessary to assign a port number to the socket because any free port is used automatically. Since the datagram packets contain the port addressing information, it doesn't matter to which port the client socket is connected. The port will be included in the datagram to the server and the server responds to that port.

```
try {
  socket = new DatagramSocket();
  // send request
```

```
  byte[] buf = new byte[8];
  InetAddress address = InetAddress.getByName(host);
  String dString="<<init>>";
  buf=dString.getBytes();
  DatagramPacket packet = new DatagramPacket(buf, buf.length, address,1500);
  socket.send(packet);
  input_thread it = new input_thread(socket);
  it.start();
}
catch (IOException e) {
  e.printStackTrace();
}
```

The variable "address" in the previous code segment contains the hostname which is generated by the string "host" which is e.g. "localhost" or "192.68.91". To create a packet of type datagram packet, four parameters are necessary. The first argument contains an array which has to be of type "byte". Therefore, a conversation from "string" to "byte" is necessary. The second parameter is just the length of the array intended to send. Finally, the server's address as well as the port number to which the server is connected to, are passed to the constructor. Here it is arbitrarily chosen port 1500.
"Socket.send(packet)" eventually sends the packet to the server. The first packet contains any string just to initiate the connection. After the first packet the server knows the address of the client and it can send packets to the client. To receive packets from the server, a thread is started to poll the socket connection for new incoming data. Since the thread has to use the socket which is initialized, the socket descriptor is passed to the thread.
A scheme of how a UDP connection works in principle, is shown in Figure 4.1.

4.2. RS232 Serial Port

The RS232 interface is a serial interface. Serial because the data sending device can just send one bit at a time. The data transfer rate reaches up to a couple of hundred kilo bauds. Depending on the speed and cable quality the wire length can be about eight meter. When the cable is longer the probability increases that the signals become unusable. Because of its popularity the RS232 port is available in many devices. In this project the serial port is used to connect the Etrax computer to one of the Atmel

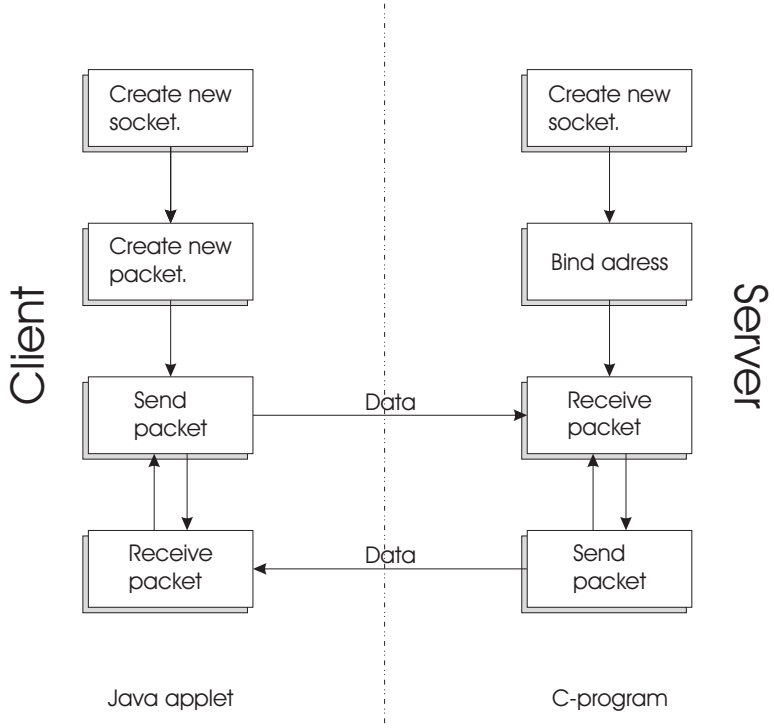

Figure 4.1.: Schematical illustration of a UDP socket connection.

controller boards. The following describes how to initialize the serial port. Also, an approach of how to send data as well as to receive data is presented.

4.2.1. Accessing the Serial Sort on the Etrax Computer

Since devices like the serial port are treated as files, it is necessary to use the "open" command to access it:

```
r2d2 = open(serial_port, O_RDWR);
```

"Serial_port" is a string defining which port to open. For instance, port "/dev/ttyS1" is used on the Etrax computer. Given that the port has to allow write accesses as well as read access, the O_RDWR flag has to be set. When the "open" command is executed, the serial port is configurable towards the serial port's file descriptor "r2d2". After the port is opened the port configuration has to be loaded and modified.

```
tcgetattr(r2d2, &settings);
settings.c_iflag = 0;
```

```
  settings.c_oflag = 0;
  settings.c_lflag = 0;
  settings.c_cflag = CLOCAL | CS8 | CREAD;
  settings.c_cc[VMIN] = 1;
  settings.c_cc[VTIME] = 0;
  cfsetispeed(&settings, B38400);
  cfsetospeed(&settings, B38400);
  tcsetattr(r2d2, TCSADRAIN, &settings);
```

The command "tcgetattr" loads the port settings in the structure "settings" which is of the type "termios", defined in the header file <termios.h>. By including <termios.h> the POSIX control functions can be used. The flags "c_iflag", "c_oflag" and "c_lflag" are set to zero. The "c_cflag" member of the terminos structure is responsible for control options. The flag CLOCAL is set. Otherwise it is possible that the serial port cannot be opened. With the set of CS8 it is defined that a datablock between the start and stop bit has a length of 8 bit. CREAD enables the serial port's receiver. The "c_cc" array contains control character definitions as well as timeout parameters. VMIN defines the minimum characters to read from the port and VTIME sets the time to wait for data in tenth of seconds.
The command "cfsetispeed" and "cfsetospeed" set the input speed to 38400 baud and the output speed respectively. With "tcsetattr" the settings are written to the serial port. The option TCSADRAIN forces the command to wait until everything is transmitted. After the initialisation the serial port is configured with the following settings: 8 data bits, 1 stop bit (since the CSTOP flag is not set) and no parity bit. Attention should be paid on the fact that the serial communication partner must have the same settings. Otherwise communication is not possible.
Now, the initialisation is complete and a receiver routine can be implemented. The receiver is programmed in a separate thread since other parts of the program would be blocked when no data is received. The file descriptor for the initialized serial port is passed to the thread:

```
void *RS232_receive(char *fd) {
  char ch;
  int i;
  while(1) {
    memset(&serial_in, 0, sizeof(serial_in));
```

```
    read(fd_,&serial_in,MESSAGE_LENGTH);
    sendUDP(serial_in);
}}
```

The array "serial_in" is used to store incoming data from the serial port. The "memset" command fills this array with zeros since a string is terminated with a zero. To terminate a string is necessary e.g. to print it out to the console.
The "read" command finally reads data from the serial port in the input array. The number of bytes which are read is defined by MESSAGE_LENGTH. Eventually, the received data is sent via the UDP socket connection to the Java-applet.
To send data to another device the write command has to be used. The write command sends MESSAGE_LENGHTH bytes of the array "out" via the serial port to another receiver:

```
void RS232_send(char* out) {
  write(fd_,out,MESSAGE_LENGTH);
  printf("sendRS232: %s\n",out);
}
```

4.2.2. Accessing the Serial Port on the Atmel Controller

In difference to the Etrax computer the serial port on the Atemga controller is not programmed through structures. In accordance with microcontroller custom it is necessary to write bits directly in registers. When the serial port is initialized it is necessary to ensure that the two serial ports which are communicating must have the same settings. Otherwise, a communication will fall through. The following code fragment shows the initialisation of the serial port:

```
void USART_Init()
{
  _CLI();
  UBRRH = 0x00; // Set baud rate
  UBRRL = 23;
  UCSRB = (1<<RXCIE)|(1<<RXEN)|(1<<TXEN);
  UCSRC = (1<<URSEL)|(3<<UCSZ0); // 8data bit, 1stop bit
  _SEI();
}
```

Before the initialisation, the global interrupt flag must be cleared to avoid a call of the interrupt routine during the initialisation. The baud rate is set by writing 23 in the UBRRL register, [7]. Then, the baud rate is set to 38400 baud at a system clock of 14.7456 MHz by normal transmission speed. Since the UBRRH register shares the I/O location with the UCSRC register, some special considerations must be taken when accessing the two registers. The most significant bit of the I/O location determines which register is selected. If this bit is set to zero the UBRRH will be written. Otherwise, the UCSRC register can be accessed.
To enable the receiver and transmitter unit the RXEN bit as well as the TXEN bit are set in the UCSRB register. The character size is determined to 8bit by writing the number three to UCSZ0. Actually, two bits are set: the UCSZ0 bit and the UCSZ1 bit. Now, the most significant bit URSEL is set to one to access the UCSRC register instead of the UBRRH register. Eventually, the global interrupt flag can be set to enable interrupts. This is done with _SEI(). After that, the UDR register can be used to receive and transmit data.
Subsequently, the procedure of receiving data is discussed. Since the point of time when data is received is undetermined the receiving of data is accomplished by an interrupt routine.

```
SIGNAL (SIG_UART_RECV)
{
...
  while (!(UCSRA & (1<<RXC)) ) ;
  temp=UDR;
...
}
```

Here just the two lines which are responsible for receiving data are of note. The rest of the interrupt routine does not belong to this chapter and will be discussed later.
The while loop does nothing as long as data is not received in UDR register. Since the interrupt is set the data must be in the UDR register. The request of the RXC bit is just for security reasons. When the RXC bit is set the data is received and can be written in "temp".
To transmit data over the RS232, two functions are necessary. The defined data protocol, explained in Section 5.1, comprises four bytes which consist of eight hexadecimal numbers. Thus, eight characters have to be sent with every single data transfer. So the

USART_puts function calls the USART_putc function 8 times:

```
void USART_puts(char *data, int length) {
  int count;
  for (count = 0; count < length; count++)
    USART_putc(*(data+count));
}
```

"Data" is a pointer to the data which will be send and "length" determines how many times the sending function is called. Eventually, USART_putc sends every single character via the serial port:

```
void USART_putc( unsigned char data )
{
  /* Wait for empty transmit buffer */
  while ( !( UCSRA & (1<<UDRE)) ) ;
  /* Put data into buffer, sends the data */
  UDR = data;
}
```

Before sending a character it has to be tested if the USART empty bit (UDRE) of the data register is set. When the bit is set the register is empty and new data can be send. Probably it is a little bit confusing that the UDR register is used to send data and to transmit data. This results out of the fact that the receiver register and the transmit register share the same address. Depending if it is a read or write access the hardware decides on its own which register is used.

4.3. Two Wire Serial Interface - TWI

The Two Wire Serial Interface (TWI) allows connecting up to 128 devices. To connect the devices only two bidirectional bus lines as well as a pull up resistor for each bus line are necessary. One line, responsible for handshaking, is called the clock (SCL). The other one transmits the data (SDA). Every single device connected to the bus has its individual address. The mechanism to resolve these addresses is inherited in the TWI protocol.
To transmit a bit via the SDA line it has to be guaranteed that the data is stable. This is accomplished by putting the clock line "high" when the data is stable. Figure 4.2 shows

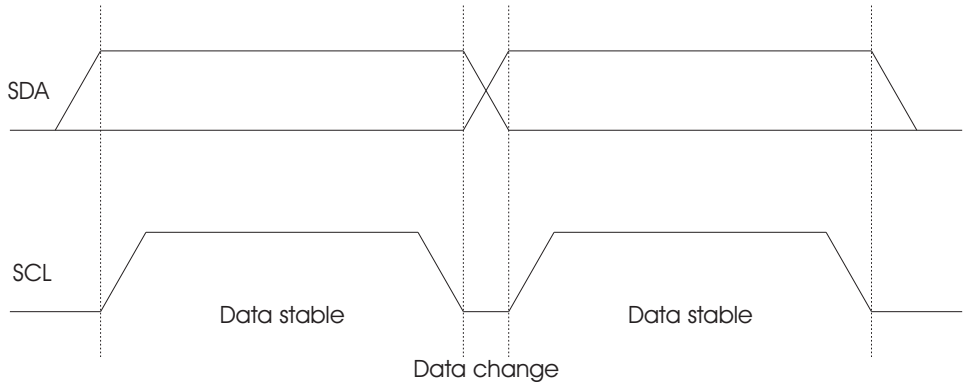

Figure 4.2.: Transmission of two bit.

the bus lines transmitting two bits. The following describes TWI connections related to the Atmega controller.

The TWI communication is based on special states. The communication between master and slave is initiated when the master issues a START condition to the bus. Then, the master is considered busy. This state of communication is determined through the status code in the TWSR register. After issuing the START condition as well as after each other action taken on the bus, the TWI hardware on the Atmega issues an interrupt. In an interrupt routine the state of the bus can be queried from the TWSR register and the next appropriate action can be taken by the software.

4.3.1. Starting and Stopping a Data Transfer

Only the master can start a data transmission. Therefore, a START condition is issued to the bus. Now, the bus is controlled by this master and considered busy. To terminate a transmission, the master generates a STOP condition on the bus. Between START and STOP a slave is addressed and data is issued to the bus. Additionally, the master can issue a new START condition between a START and a STOP. This is called a REPEATED START condition. A REPEATED START condition is used when the master finished the transmission of data and does not want to loose control of the bus. A START condition is detected when a falling edge occurs on the SDA line by ”high” level on the SCL line. A rising edge on the SDA line by ”high” SCL line is associated with a STOP condition. Figure 4.3 illustrates the discussed conditions.

4.3.2. Address and Data Packet Format

After a START condition the master addresses the slave by sending a nine bit address packet to the bus. The highest seven bits contain the slave's address followed by a READ/WRITE bit. This bit determines if the master wants to write data or to receive data from a slave. If this bit is set, a read operation is performed. In case of a zero, the master wants to send data to the slave. The last bit is an acknowledge bit. If the address packet is successfully read by the slave, the SDA line is pulled down to acknowledge the address transmission. In case the slave did not acknowledge the address, the master can send a STOP condition or a REPEATED START condition to start a new transmission. The slave's address can be chosen freely. Excluded, however, is the address 0000 000 since this address is reserved for a general call. By issuing a general call all slaves are asked to respond. A general call in conjunction with a WRITE bit causes that the transmitted data sent by the master is received by all acknowledging slaves. Furthermore, all addresses 1111 xxx are reserved for future purposes. An address packet is depicted in Figure 4.4.

After the slave has acknowledged the address packet, the master sends the data packet. The data packet consists of eight data bits followed by an acknowledge bit. The receiver issues an acknowledge by pulling the SDA line down in the ninth clock cycle. If this is not accomplished, a "not acknowledge" (NACK) is issued. This should be done to signal that the last data byte is received.

It should be mentioned that a master as well as the slave can be both receiver or transmitter. Figure 4.5 shows the format of a data packet.

In general a transmission begins with a START condition followed by the address packet. Then the data packet is issued with an arbitrary number of data bytes. The transmission is ended with a STOP condition generated by the master. Figure 4.6 shows a scheme of a typical data transmission.

All clock lines of the devices connected to the bus are wired-AND. This has the advantage that the slave can extend the "low" period of the SLC line. For instance, if the master's clock is too fast, the slave extends the "low" period of the clock to have time enough to read the data line. The clock period which is determined by the master is not affected by this.

It is possible to have more than one master connected to the bus. Since this is not used in this work, refer to [6] or [7] to obtain deeper information about this topic.

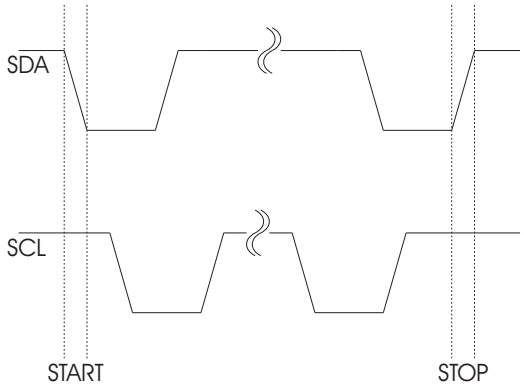

Figure 4.3.: START and STOP condition.

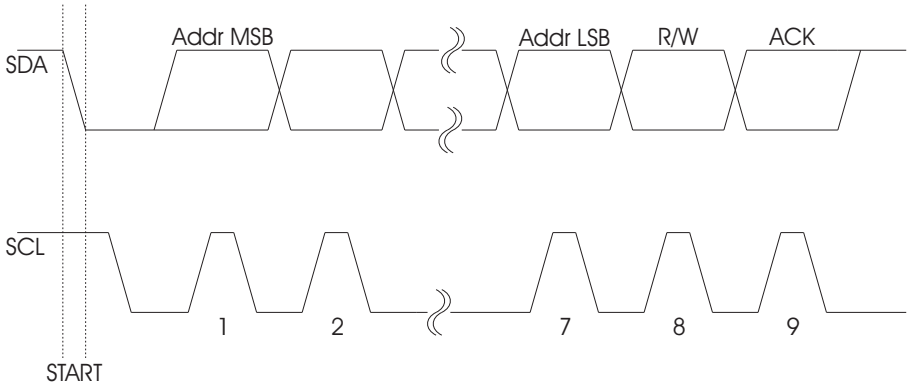

Figure 4.4.: Address packet format

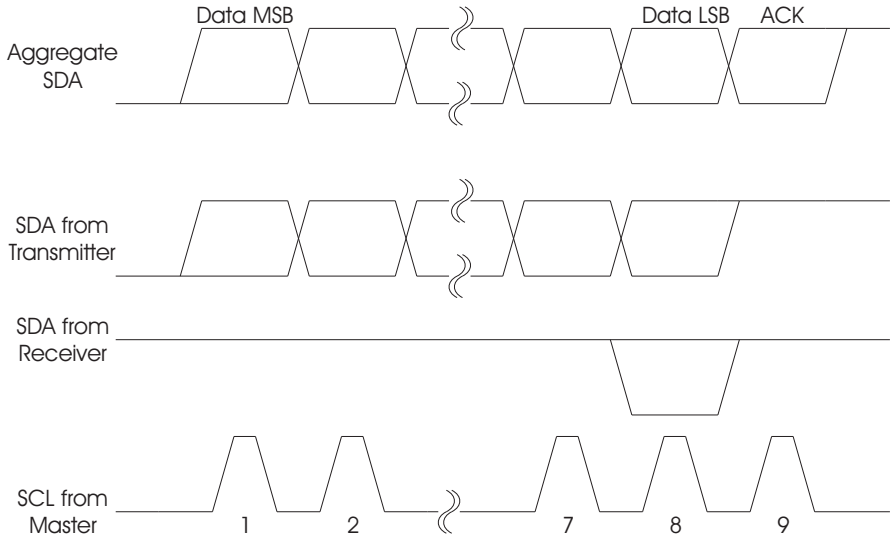

Figure 4.5.: Data packet format

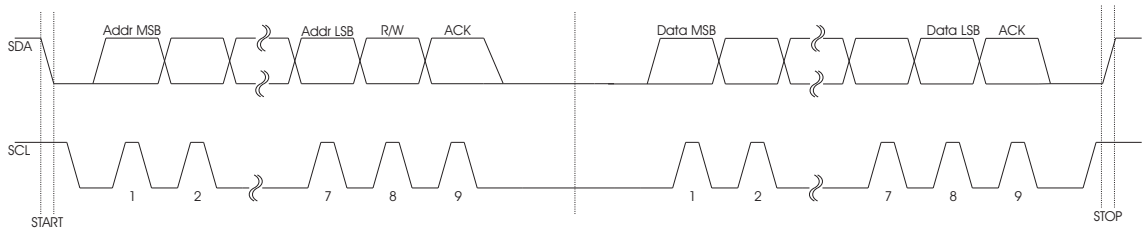

Figure 4.6.: Typical data transmission

4.3.3. Transmission Modes

Four major modes are available in which a device can operate: Master Transmitter, Master Receiver, Slave Transmitter and Slave Receiver. These modes can be chosen by the software and the user has to take care that a capable mode is selected to achieve a smooth data transfer.

4.3.4. Master Transmitter and Slave Receiver

The Atmega16 controller receives motor commands from the Etrax computer. These commands must be forwarded to the Atmega8 via the TWI bus. To accomplish a data transfer, the Atmega16 operates in master mode and the Atmega8 in slave mode, respectively. In the case of this application, the master sends four bytes to the slave. These four bytes e.g. contain values to set the motor speed of the robot drives. The protocol used to exchange information between the robot's components is explained in Section 5.1.
The procedure of accomplishing a data transmission from master to slave is depicted in Figure 4.7 and 4.8. First, the master issues a START condition by writing the TWINT and TWEA bit. Thus, an interrupt occurs in the master and the status code $08 is written to the TWSR register. By setting the TWINT bit, the next action taken by the TWI hardware is transmitting the slave address and waiting for "acknowledge" or "not acknowledge". When the slave successfully received its address, it transmits "acknowledge" to the bus which results in an interrupt and the slave status code $60. Setting bit TWINT and TWEA forces the slave to wait for data. At the same time but after the slave sent "acknowledge", the master jumps into the interrupt routine and the status code $18 is written to the TWSR register. By setting the TWINT bit, the master transmits the data and waits for "acknowledge". Now, the slave receives the sent data which results in sending "acknowledge" to the bus and the status code $80 appears in

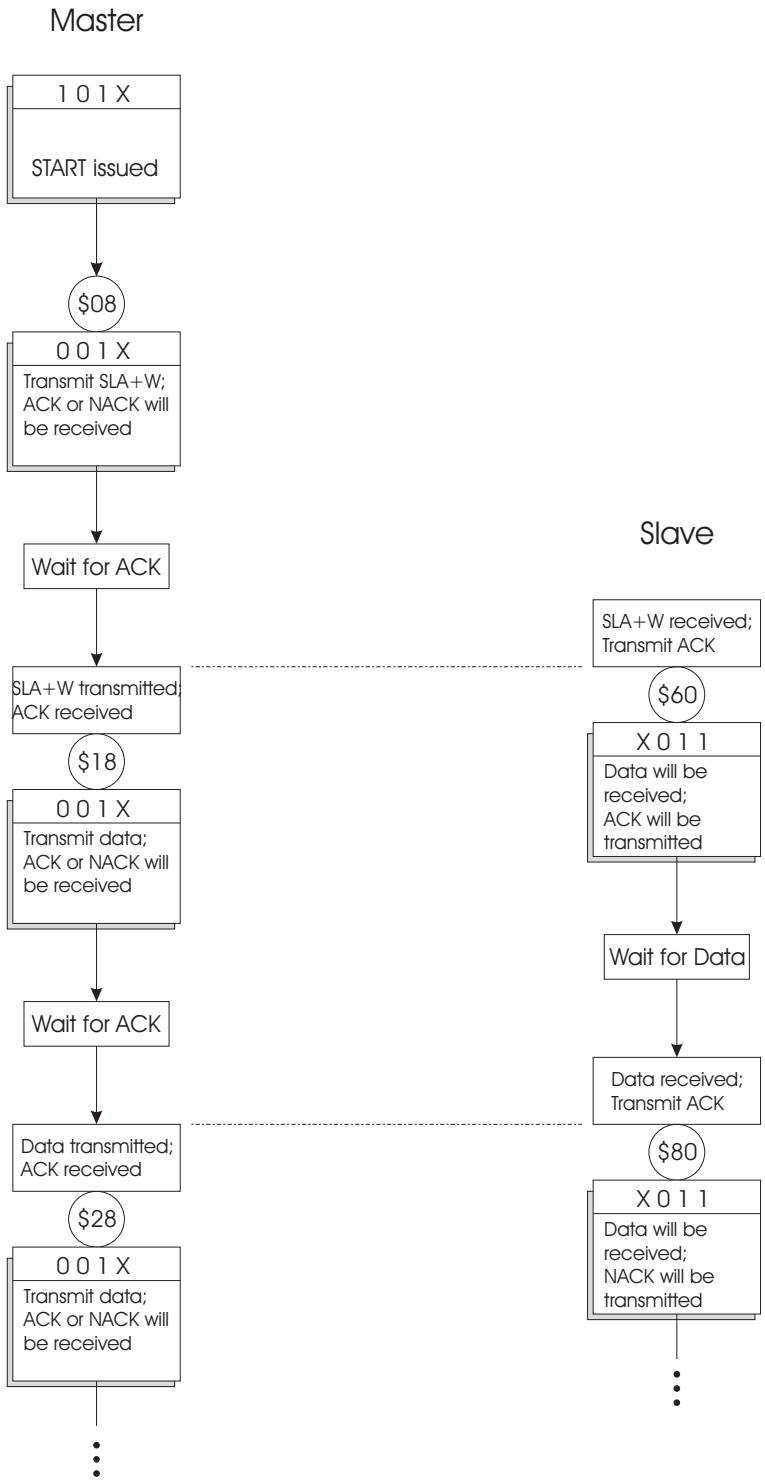

Figure 4.7.: Interaction of master and slave to send a data packet of four byte - part 1

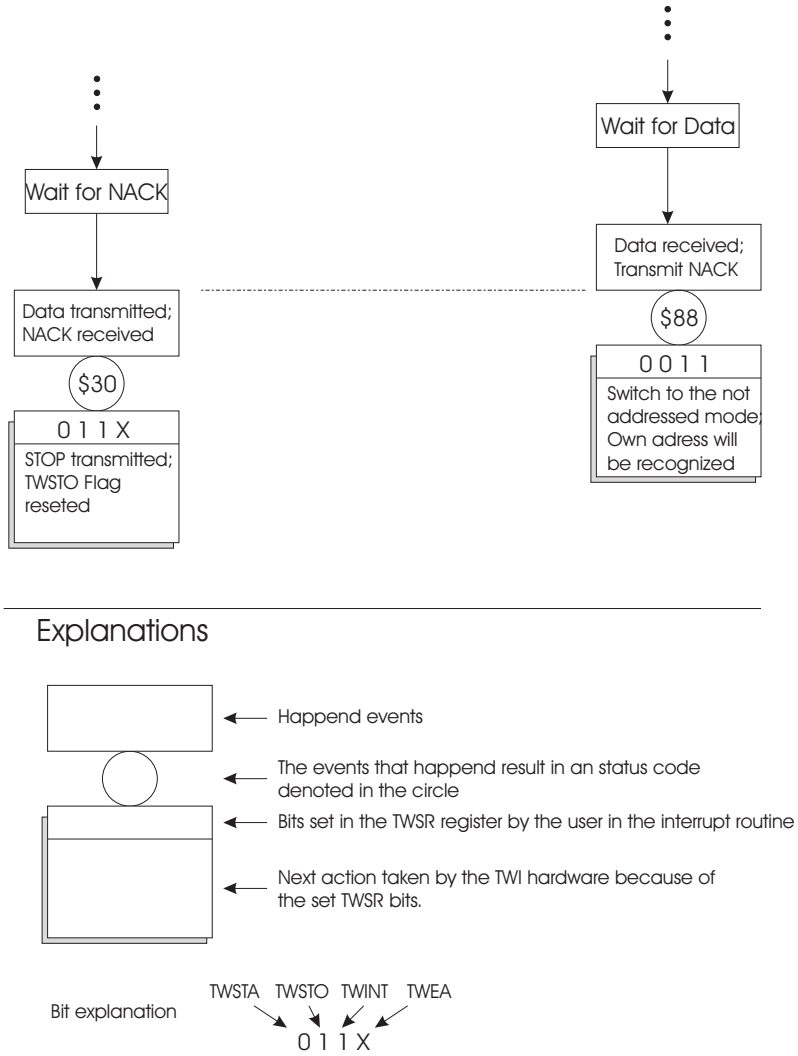

Figure 4.8.: Interaction of master and slave to send a data packet of four byte - part 2

the TWSR register on the slave. After setting the bits TWINT and TWEA, the slave waits for data again. Meanwhile, the master receives "acknowledge". The status code is now $28. Since the master has to transmit the data and to wait for "acknowledge", the TWINT bit is set. After receiving the data, the slave jumps into the interrupt routine again. The status code is $80. Now, this procedure repeats until the fourth byte is transmitted by the master. Then, the slave issues a "not acknowledge" to the bus which results in the slave status code $88. Now, the slave switches into not addressed mode. The master transmits a stop condition to the bus. Thus, the transmission of four bytes is finished and a new packet of four bytes can be sent by issuing a new START condition. Remark: It has to be taken into account that every time the TWCR register is written, the TWEN bit has to be set to enable the TWI interrupt. Another illustration of the states which are the master and the slave are in as well as the action which has to be taken in the interrupt routine, is given in [6] or [7].

Atmega16 - Master Transmits

The following describes the implementation of the Master Transmition mode on the Atmega16 controller.
First, it is necessary to initialize the TWI hardware. The value in the TWBR selects the division factor for the bit rate generator. TWSR is responsible for the prescaler of the bit rate generator. A table of possible values is given in [6] or [7]. The following code segment shows the initialization method:

```
void TWI_Init() {
  _CLI();
  TWBR = 0x0a;
  TWSR = (0<<TWPS1) | (0<<TWPS0);
  _SEI();
}
```

The four bytes which will be sent by the master have to be available in the array "dataT". To write the data to the array, the method TWI_transmi_4Byte is used. It splits a four byte integer value into four single bytes and writes it to the array "dataT"'. Having done this, the "read" flag is set to zero and the START condition is issued in the last line of the following code segment. The "read" flag determines if the master sends data to the slave or requests information from the slave.

```
void TWI_transmit_4Byte(uint32_t hex)
{
  int i=0;
  for (i=3;i>-1;i--) {
    dataT[i]=hex;
    hex = hex >> 8;
  }
  read = 0;
  TWCR = (1<<TWSTA) | (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
}
```

When the bus changes its state, an interrupt occurs in the Atmega controller and a status code is written to the TWSR register. To get the status code, the lowest two bits have to be masked since they contain the value of the prescaler for the bit rate generator.

```
SIGNAL(SIG_2WIRE_SERIAL) {
  switch (TWSR & 0xf8)
    {
    ...
```

Status code 0x08 in the status register indicates that START condition has been transmitted. Therefore, the address is written to the TWDR register. Since the "read" flag is zero the lowest bit is set to zero the mode. This results in entering the Master Transmission mode. Now, the next action is accomplished by setting the appropriate bits in the TWCR register.

```
    case 0x08:
      {
        if (read==0)
          TWDR = 0x44;
        else
          TWDR = 0x45; // read from slave
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
        flagT=0;
        flagR=0;
        break;
      } ...
```

If the slave receives its address successfully, "acknowledge" is issued. This results in the status code 0x18. Then the first data byte can be sent. TWCR is written to send the data in TWDR and wait for the slave acknowledging the data.

```
    case 0x18:
      {
      TWDR = dataT[flagT];
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
      break;
      }
      ...
```

When the slave acknowledged the reception of the data, TWSR contains the status code 0x28. Then, the next byte can be sent and the master waits for acknowledgement.

```
    case 0x28:
      {
      flagT++;
      TWDR = dataT[flagT];
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
      break;
        ...
      }
```

This procedure is repeated until the slave quits with sending "not acknowledge". Then, the status register contains 0x30. That indicates that the transmission of four bytes has finished. Accordingly, a STOP condition has to be issued:

```
    case 0x30:
      {
      TWCR = (1<<TWSTO) | (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
      break;
      }
    }
}
```

Atmega8 - Slave Receives

Before it is possible to communicate over the TWI bus, the slave has to be initialized. In the address register TWAR, the slave address is set. This address must be identical with the address assumed in the master. If it is not the same address, the slave will not recognize its own address on the bus. The macro _CLI() disables the global interrupt flag and _SEI() sets it again. To initialise the slave, the following method is used:

```
void TWI_init() {
  _CLI();
  TWAR = 0x44;
  TWBR = 0x0a;
  TWSR = (0<<TWPS1) | (0<<TWPS0);
  TWCR = (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
  _SEI();
}
```

If the state changes on the bus, also an interrupt routine is invoked in the slave controller. But the status codes are different to the master's codes. After the master has transmitted the slave's address and the address has been received by the slave, the status code saved in the TWSR register is 0x60. In this case, the TWI hardware has to wait for the first data byte. Therefore, TWINT and TWEA has to be set in the control register TWCR:

```
SIGNAL(SIG_2WIRE_SERIAL) {
  switch (TWSR & 0xf8)
    {
    case 0x60:
      {
      flagR=0;
      TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
      break;
      }
      ...
```

When the first data byte is received, the status code 0x80 appears in the status register TWSR. Now, the first received data byte is stored in the array "dataR". After this, the counter "flagR" is increased. As long as all four data bytes are not received, the TWI

hardware waits for another data byte. If "flagR" reaches the value of three, the fourth data byte is receipt of putting a "not acknowledge" out of the bus:

```
case 0x80:
  {
  dataR[flagR] = TWDR;
  flagR++;
  if (flagR<3) {
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
  } else {
    TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
  }
  break;
  }
  ...
```

When the last data byte is eventually received and "not acknowledge" has been returned, the status register contains the value 0x88. Then, the last byte is stored in the array "dataR". The next code lines write the four bytes from the array into a four byte integer number. After that, the selection of the channel follows which is described in the section 5.2. At the end it is switched to the "not addressed" mode by setting the bits TWINT and TWEA in the TWCR register.

```
case 0x88:
  {
  dataR[flagR]=TWDR;

  uint32_t hx=0;
  int i=0;
  for (i = 0 ; i < 4  ; i++) {
    hx = hx << 8;
    hx |= dataR[i];
  }
    ...
    // described in another section
    ...
```

```
        TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
        break;
      }
    }
}
```

4.3.5. Master Receives and Slave Transmits

To ensure a bidirectional data transfer, the master also has to be able to receive data. Then, the master operates as the receiver and the slave as the transmitter. In this operation mode the transmission is also initiated by the master. Thus, the master can retrieve information from the slave but the slave cannot start a data transmission.

Atmega16 - Master, Receiver

To retrieve information from the slave, the master has to issue a START condition to the bus:

```
read = 1;
TWCR = (1<<TWSTA) | (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
```

When the START condition is successfully issued, the status register contains the value $08. Since the "read" flag was set to one, the LSB bit in the address packet is now set to induce a data request from the slave:

```
...
case 0x08:
  {
    if (read==0)
      TWDR = 0x44;
    else
      TWDR = 0x45; // read from slave
    TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    flagT=0; flagR=0;
    break;
  }
...
```

After the slave has acknowledged the reception of the address packet, the status register TWSR contains the status code $40. To wait for data, the bits TWINT and TWEA are set in the control register:

```
...
case 0x40:
  {
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    break;
  }
...
```

When the first data byte is received, the switch instruction jumps to case $50 and saves the received byte to the array "dataR". Simultaneously, the master acknowledges the reception of the byte. Since three data bytes are received, the switch instruction jumps three times to this case. The fourth time the "if" instruction is entered, "flagR" is equal to three and thus the "else" branch of the "if" instruction is entered:

```
...
case 0x50:
  {
    dataR[flagR]=TWDR;
    flagR++;
    if (flagR<3) {
      TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    } else {
      TWCR = (1f<TWINT) | (1<<TWEN) | (1<<TWIE);
    }
    break;
  }
...
```

In the "else" branch a "not acknowledge" is issued to the bus. After the reception of the last data byte, the status register contains the status code $58. Now, the last byte is stored and a STOP condition issued to the bus:

```
...
case 0x58:
  {
    dataR[flagR]=TWDR;
    TWCR = (1<<TWSTO) | (1<<TWINT)| (1<<TWEN) | (1<<TWIE); // stop
    break;
  }
}
```

Atmega8 - Slave Transmits

The LSB of the address packet indicates that the slave is asked to send data to the master. That results in the status code $A8. To transmit the first data byte and wait for the acknowledgement of the master, TWINT and TWEA is written to the control register TWCR:

```
...
case 0xA8:
  {
    flagT=0;
    TWDR=dataT[flagT];
    flagT++;
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    break;
  } ...
```

When the "acknowledge" is received (status register contains 0xB8), a further byte is transmitted. After the acknowledgement, this "case" is entered again. This is repeated until three data bytes are send. Then, the "else" branch of the "if" instruction induces the reception of a "not acknowledge":

```
...
case 0xB8:
  {
    TWDR=dataT[flagT];
    flagT++;
    if (flagT<4) {
```

```
      TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    } else {
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    }
    break;
  } ...
```

Is the "not acknowledge" received, the status register contains the status code $C0. In this case, the slave switches to "not addressed" mode:

```
...
case 0xC0:
  {
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    break;
  }
}
```

4.4. Multithreaded Programming

Here multithreaded programming or concurrent programming refers to multiprogramming parallelism. That means that threads are executed on a single processor so that the parallelism is achieved by multiplexing the threads. Therefore, true parallelism where every thread runs on its own processor is not possible. But in case of this work the logical pseudo parallelism is sufficient.
On the Etrax computer the "pthread" library has to be installed in order to be able to multi-threading. Otherwise, a multithreaded program will not run on the Etrax computer.
Since the Java-applet and the C-program running on the Etrax computer uses threads, it is explained in the following sections how to implement threads in C as well as in Java.

4.4.1. Threads Implemented in C

Threads in C are created by using the "pthread_create" function. The prototype of the function "pthread_create" can be stated as follows:

```
int pthread_create(pthread_t *threadp, const pthread_attr_t ...
        ... *attr, void* (*start_routine)(void *), arg *arg);
```

The function needs an address of the "pthread" object which has to be declared before. The second parameter can be used to modify the threads attributes. By passing a NULL to the function the standard settings are used. "Start_routine" is the name of the thread function which contains the threads program code. The last parameter to pass to "pthread_create" is an argument e.g. an address of a variable or array which is used in the thread. When the thread is successfully completed a null is returned. In the case of an error the function returns any other value. To wait for the termination of a thread the "pthread_join" function has to be used:

```
int pthread_join(thread_t threadp, void **status);
```

This function forces the calling unit, in our case the main program, to wait for termination of the running thread. The name of the "pthread" object which the calling program has to wait for has to be passed. The second argument points to a location that is set to the exit status of the terminated thread but here this argument will always be NULL. As mentioned the "pthread_create" function needs the address of a "start_routine" which is the name of the actual thread function. This thread function is declared as follows:

```
void *<name of the thread>(type  *<variablename>) {
}
```

If in the thread function an endless loop is implemented the thread never ends and so the "pthread_join" command waits endlessly for the thread termination.
The explanations in this section comprises by far not everything what is possible with threads in C but it is enough for the work at hand.

4.4.2. Threads Implemented in Java

There are two different ways to create a thread in Java. One possibility is to extend a class from the "thread" class. Another way is to implement the class as "runnable". This is advantageous when the class needs also to extend another class than a thread. Since the extension of another class is not necessary the first way of creating a thread class is used. For further information about the second way refer to [5].
Creating a thread by extending the from the "thread" class works as shown below:

```
public class MyThread extends Thread {
  public void run() {
    // Code to be executed
  }
}
```

To start the thread an instance of the class "MyThread" has to be created and the start method has to be invoked:

```
MyThread m = new MyThread();
m.start();
```

Now, a problem consists in passing parameters to the thread. One solution would be to define the "MyThread" class as an inner class of the main program. But this is not possible in Java-applets. Another way is to pass parameters with the constructor to the thread. For instance, an integer is passed to the thread:

```
public class MyThread extends Thread {
int number;
  MyThread(int number) {
    this.number=number;
  }

  public void run() {
    // Code to be executed
  }
}
```

Then the main program creates and starts the thread as follows:

```
...
int number;
number=10;
MyThread m = new MyThread(number);
m.start();
...
```

5. System Assembly

In the previous chapters all single components of the robot have been described. In this chapter the objective is to point out how the components work together. Figure 5.1 shows the block diagram of the whole system. The host computer on the right side connects the internet to the local area network which belongs to the robot. All components left from the host belong to this local network. In the center two wireless LAN adapters can be seen. They form the connection between the mobile parts on the left side-the components which are mounted on the robot- and the stationary parts on the right side.
The network camera is connected to the left wireless LAN adapter. Also connected to this wireless LAN adapter is the Etrax computer. The Etrax computer communicates over its RS232 serial port to the Atmega16 controller. This controller is responsible for reading sensory data from the bumper switches. Also, it is responsible for forwarding data between the Etrax computer and the Atmega8 controller. The Atmega8 controller implements all necessary functions to control the robot's motors. The communication between the two Atmega controllers is accomplished by the serial TWI bus. To this bus are also connected the controllers of the robot's arm. But this is not part of this thesis. The following describes how information is exchanged between the robot components. Therefore, it is necessary to define a protocol.

5.1. Communication Protocol between the Components

If the Atmega16 controller recognizes a collision between the robot and an obstacle the Atmega16 AVR has to forward this "event" to the Atmega8 controller. Then, the Atmega8 controller can stop or control the motors to avoid harm. Since most components have to work together like the two Atmega controllers, it must be possible to exchange data. This is accomplished with data packets send from one device to another. The packet contains an address to which device the package has to be sent. Further, a command is contained to specify an action to be performed. The last component of a packet

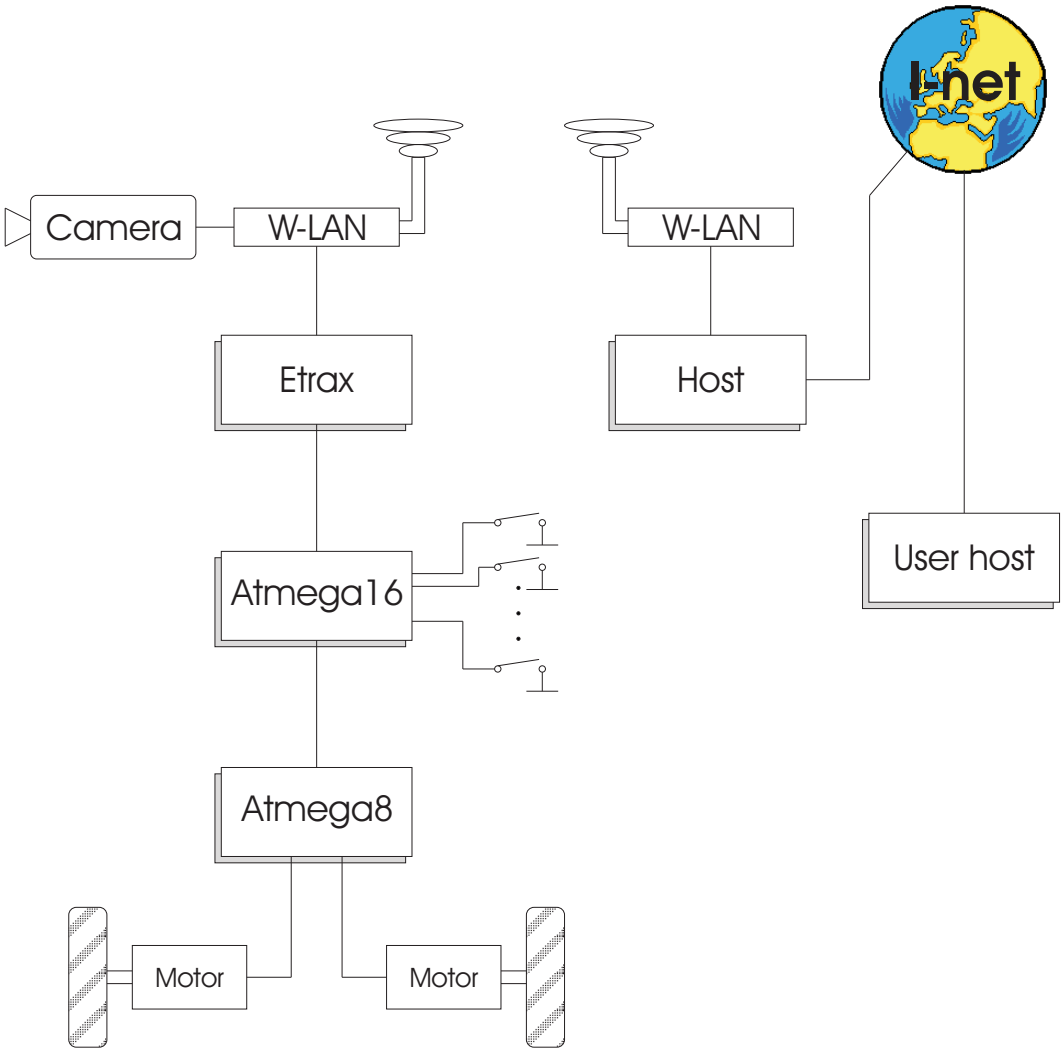

Figure 5.1.: This diagram illustrates how the robot's components are put together and how these components work together with external devices.

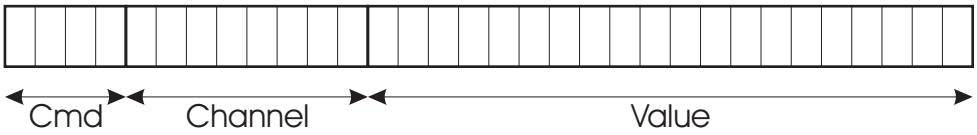

Figure 5.2.: Format of the message packet used for information exchange between the components.

is a value.
The mentioned three components of the packet have to be put in a protocol which has to be defined. This protocol determines in which manner data can be transferred and interpreted. It is assumed that a device needs to have the capability to set values and read values of other devices. To define these commands, four bits are reserved. This makes it possible to define 16 different commands. Because of these commands a device knows how the data to proceed but it is not determined yet to which device the command should be delivered. To define the device to which the packet is addressed to, eight bits are reserved in the packet. By setting these eight bits a channel is determined to which the command belongs to. A channel is not necessarily a hardware device. More than one channel can be inherited in one hardware component. E.g. the motor controlling Atmega8 controller has at least two channels: One channel for the left motor and one channel for the right motor. The value takes 20 bits in a packet. So the whole message packet comprises four byte. The first four bit are used for the command, the next eight bit determine the channel to which the packet has to be sent and the last 20 bits represent the value. Figure 5.2 illustrates the defined message packet.
Remark: Even if a packet contains four byte of information, eight byte are send. This results from the fact that the interfaces just can send characters. One character represents one hex number and four byte consist of eight hexadecimal numbers.

5.2. Channel and Command Selection

When a message packet reaches the Atmega16 it has to make a decision to which device it sends the packet or if the packet is determined for itself. The following code segment is located in the USART interrupt routine. When a data packet is received and stored in the array "ch", then it is converted to a 32 bit long integer named "hex". The function "convertStringToHex" is described in Section 5.5.

```
...
  hex=convertStringToHex(ch);
...
```

Now, the channel, the command as well as the value with its sign are extracted out of the variable "hex":

```
...
  cmd = (hex & 0xf0000000)>>28;
  chan = (hex & 0x0ff00000)>>20;
  sign = (hex & 0x000f0000)>>16;
  value = hex & 0x0000ffff;

  switch (chan)
    {
      case 8: Ch8selectCmdAndSendData(cmd,sign,value);
      default: TWI_transmit_4Byte(hex); break;
    }
...
```

The function "Ch8selectCmdAndSendData" takes received data and selects the command to be executed. This function assumes that the selected channel is situated on the current device. If the message packet contains data for the other channels, the message packet is send further to another Atmega controller by using the function TWI_transmit_4Byte in the "default" case condition.

5.3. Information Exchange between the Java-Applet and the Etrax Computer

5.3.1. Sending data from the Java-applet to the Etrax-computer

The Java-applet sends its commands via UDP socket connection to the Etrax computer. The Etrax computer does not have to process the received data. It just has to forward it. To send a message packet the command number, the channel number and the value are put together in a string containing four bytes of information. To achieve this, a function is written which puts the three components together to a message packet. The function provides the possibility to pass a sign which represents the sign of the value.

If the sign is one, the value is negative. Otherwise, it is positive. The first lines of the function are as follows:

```
public void SendMessageViaUdp(String Cmd, String Channel, ...
                                            ...int Value, int sign) {
    String CmdChannelValue=Cmd;
    String strValue;

    CmdChannelValue = CmdChannelValue.concat(Channel);
...
```

The variable "CmdChannelValue" represents the string which contains the whole message packet. One advantage that makes the handling of strings easy is that strings in Java do not have to have fixed lengths. With the declaration of "CmdChannelValue" the command is copied to the string. The last line of the above code segment concatenates the channel-number to the message packet. Now, the packet contains the command number followed by the channel number. Since the value is an integer number, it has to be converted to a string:

```
...
    strValue = Integer.toString(Value,16);
...
```

The string "strValue" contains the string representing the value. The number "16" specifies the enumerative system to which the value is converted to. Since it is necessary to send hexadecimal numbers, the "16" specifies the hexadecimal number system.
Depending on the length of the hex string, zeros have to be padded to the message packet to have always a packet length of eight hexadecimal numbers:

```
  ...
    if (sign==1) {
        switch (strValue.length())
        {
        case 1: CmdChannelValue=CmdChannelValue.concat("8000"); break;
        case 2: CmdChannelValue=CmdChannelValue.concat("800"); break;
        case 3: CmdChannelValue=CmdChannelValue.concat("80"); break;
        case 4: CmdChannelValue=CmdChannelValue.concat("8"); break;
```

```
        }
    }
    else
        {
        switch (strValue.length())
            {
            case 1: CmdChannelValue=CmdChannelValue.concat("0000"); break;
            case 2: CmdChannelValue=CmdChannelValue.concat("000"); break;
            case 3: CmdChannelValue=CmdChannelValue.concat("00"); break;
            case 4: CmdChannelValue=CmdChannelValue.concat("0"); break;
            }
        }
...
```

When the "sign" parameter indicates that the value is negative, the first bit of the 20 bit long value is set to one. This bit is used as the sign bit. After the message packet is padded with zeros and the sign bit is added, the value can be concatenated:

```
...
    CmdChannelValue = CmdChannelValue.concat(strValue);
...
```

Now, the string "CmdChannelValue" contains the whole message packet. It can be sent via the UDP socket connection to the Etrax computer:

```
...
        byte[] buf = new byte[8];
        buf = CmdChannelValue.getBytes();
    try {
        InetAddress address = InetAddress.getByName(host);
        DatagramPacket packet = new DatagramPacket(buf, ...
                                        ... buf.length, address, 1500);
        socket.send(packet);
    }
    catch (IOException e) {
        e.printStackTrace();
    }}
```

For explanations to the last code segment it is referred to Chapter 4.1.2 where the UDP socket connection is described.
The sent data has to be received from the C-server running on the Etrax-computer. Therefore, the thread "receiveUDP", described in Section 4.1.1, is used.

5.3.2. Sending Data from The Etrax Computer to the Java-Applet

To send data to the Java-applet, the function "sendUDP" is used:

```
void sendUDP(char *data) {
  n = sendto(sd, data, MESSAGE_LENGTH, 0, ...
        ... (struct sockaddr *) &cliAddr, cliLen);
}
```

For explanations to this function refer to Section 4.1.1. To receive the data that is sent from the Etrax computer, a thread is used.

5.4. Information Exchange between the Etrax Computer and the Atmega16 Controller

5.4.1. Sending Data from the Etrax to the Atmega16

The Etrax computer and the Atmega16 controller exchange data via the RS232 serial port. The C-server on the Etrax computer uses the method "RS232_send" (Section 4.2.1) to send the four byte message packet to the Atmega16:

```
void RS232_send(char* out) {
  write(fd_,out,MESSAGE_LENGTH);
...
}
```

The Atmega16 receives the sent data with the help of an interrupt routine handling incoming data from the serial port, Section 4.2.2.

5.4.2. Sending Data from the Atmega16 to the Etrax

The function on the Atmega microcontroller board used to make a string out of numbers accomplishes the same as the Java function discussed before. The function is passed a

32 bit value. Since the message packet contains a value of 20 bits, the user has to take care that this value is not longer than 20 bit. Otherwise, the higher bits will be cut off and not transmitted:

```
char* makeString(char *cmd, char *channel, uint32_t value)
{
  static char sendData[8];
  char hex[8];
  sprintf(hex,"%lx",value);
...
```

The array "sendData" of datatype char, contains the message packet which is composed of the command number, the channel number and the value. "Sprintf" copies the hexadecimal numbers in "value" to the array "hex". Thus, "value" is now accessible as a string through the array "hex". The first sequence the message packet contains is the command. So it has to be copied in the packet. Furthermore, the channel number is concatenated:

```
...
  strcpy(sendData,cmd);
  strcat(sendData,channel);
...
```

Since the message packet contains eight hexadecimal numbers, the packet has to be padded with zeros depending on the size of the "value":

```
...
  if (value < 17)
    { strcat(sendData,"0000"); }
  if (value > 16 && value < 256)
    { strcat(sendData,"000"); }
  if (value > 255 && value < 4096)
    strcat(sendData,"00");
  if (value > 4095)
    { strcat(sendData,"0"); }
  strcat(sendData,hex);
  return sendData;
}
```

Eventually, the value is added to the packet and the message packet is given back as a string. Then, the function "USART_puts", decribed in Section 4.2.2, is used to send the data.
To receive data on the Etrax computer, the thread "RS232_receive" is used - see Section 4.2.1.
Remark: Later in this work it turned out that this function had to be replaced by an more storage saving algorithm.

5.5. Information Exchange between the Atmega16 and the Atmega8

5.5.1. Sending Data from the Atmega16 to the Atmega8

After receiving the data from the Etrax computer, the channel is selected to which controller the data is determined for (refer to Section 5.2).
If the data is determined for the Atmega8 controller, the data is passed via the TWI bus. Therefore, the function "TWI_transmit_4Byte" is used. This function is described in Section 4.3.4. The Atmega16 operates as the master and its task is to retrieve information from the Atmega8 controller since the Atmega8 cannot initiate a data transfer by itself.
The Atmega8 which operates in slave mode, receives the data by handling the interrupt and the status codes. This is explained in Section 4.3.4. Since the received data packet is a string of eight characters, it is necessary to convert this string to a four byte number. This is accomplished by the function "convertStringToHex". It expects a pointer to the string and returns a four byte integer number:

```
uint32_t convertStringToHex(char *string) {
  uint32_t i=0,fourbytes=0,value=0;
...
```

A loop treats every single character and converts it to the appropriate number:

```
...
  for (i=0;i<MESSAGE_LENGTH;i++)
    {
      if ('0' <= *(string+i) && *(string+i) <='9') {
    value = *(string+i) - '0';
```

```
      }
      else if (('A' <= *(string+i) && *(string+i) <= 'F')) {
    value = *(string+i) - 'A' + 10;
      }
      else if (('a' <= *(string+i) && *(string+i) <= 'f')) {
          value = *(string+i) - 'a' + 10;
      }
...
```

After converting a character, the value of the character is added to a four byte number:

```
...
      fourbytes = fourbytes << 4;
      fourbytes |= value;
    }
return fourbytes;
}
```

After the loop has finished, the number "fourbytes" contains the converted string. For instance, if the string contained the characters AF65B8FC, the number "fourbytes" has the value 0xAF65B8FC after the conversation.

5.5.2. Sending Data from the Atmega8 to the Atmega16

If for instance status information is retrieved by the Atmega16 controller, a TWI data transfer is initiated by the Atmega16. The Atmega8 controller operates as the slave and sends the data packets by using the interrupt routine described in Section 4.3.5. To compose the data packet out of numbers the function "makeString" is available. It is the same function as on the Atmega16 controller.
The data is received by the Atmega16 also with an interrupt routine. That routine is described in Section 4.3.5.

5.6. Pulse Width Modulation to Drive the Motors

The Atmega8 drives the robot motors. In order to realize different speeds, pulse width modulation is used. On the Atmega8 two PWM modes are available. Here the "Phase and Frequency Correct PWM Mode" is used. This mode provides a phase and frequency

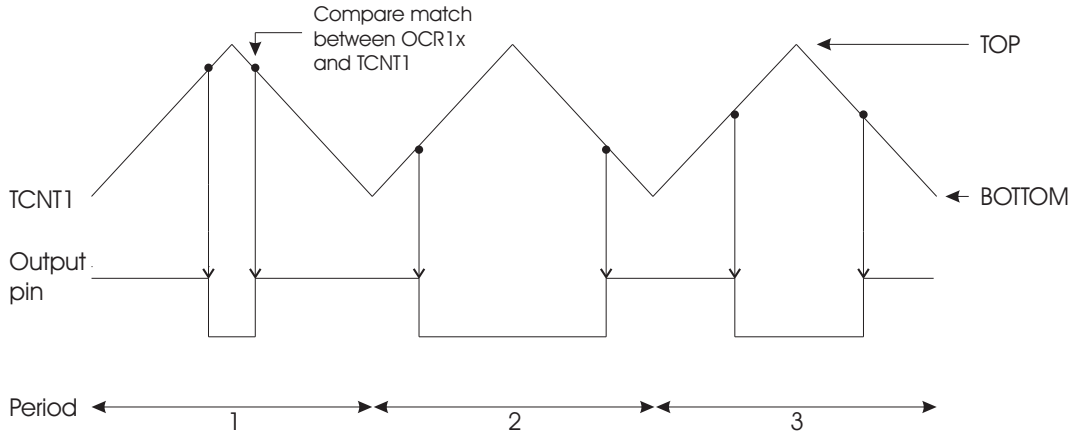

Figure 5.3.: Timing diagram of the Phase and Frequency Correct PWM mode. The timer value is shown as a histogram. The TOP value is assumed constant. If the OCR1x registers are written, they are updatet when the timer reached the BOTTOM value.

correct waveform generation and is based on dual-slope operation.
The counter TCNT1 counts from the BOTTOM (0x0000) to the TOP value. Since "non inverting output compare mode" is used the output pins are cleared every time the OCR1X registers matches the timer TCNT1 while up counting and set while down counting. Figure 5.3 shows the timing diagram. Before a usage of the PWM unit is possible, it has to be initialized.
To clear the output pins while up counting and set while down counting, the bits COM1A1 and COM1B1 have to be set in the TCCR1A register. The WGM13 bit in the TCCC1B register is set to select the PWM Phase and Frequency Correct mode. To set the prescaler for the timer speed, the bits CS10 to CS12 are set which means no prescaling. The TOIE1 bit in the TIMSK register is set to enable the interrupt on timer overflow. In the selected PWM Phase and Frequency Correct mode, the ICR1 register is used to determine the TOP value of the counter.

```
void PWM_Init()
{
  _CLI();
  ICR1=0x04FF;
  TCCR1A = 0xa0;  // (1<< COM1A1)|(1<< COM1B1)
  TCCR1B = 0x11;  // (1<<WGM13)|(1<<CS10)
  TIMSK = 0x04;
```

```
  _SEI();
}
```

When the PWM unit is initialized, the motor speed can be set by changing the value of the OCR1A and the OCR1B register respectively. This is done in the USART interrupt routine by calling the functions "Ch0selectCmdAndSendData" and "Ch1selectCmdAnd SendData" when new motor control commands are received.

The motors are designed for 6V. Since the output of the motor drivers deliver 12V, it should concidered that the pulse-duty factor of the PWM is not higher than 50 percent. In case this value is exceeded, the motor current will be too high so that the motor drivers will switch off the motors.

6. Remote Control - Teleoperation

To realise a remote control of the mobile robot, a Java-applet is written. This applet connects to the server socket running on the Etrax computer. The Java-applet as well as an html file are located on the Etrax' web server. It is possible to download the html file from any computer which has access to the internet. The Java-applet is started by the html file and through the appearing control panel the robot can be steered. For orientation issues, a video stream taken by the network camera can be seen above the control panel.
In the following, the Java-applet's functions are described.

6.1. Generating Motor Commands from the Java Control Panel

On top of the web site, an image of the network camera can be seen and below the Java control panel is started. The control panel involves a simple field to steer the robot - Figure 6.1. The robot's velocity and alignment are determined by the mouse pointer. When pointing the mouse to the origin of the field, the velocity is set to zero and no direction is determined. If the mouse moves in the upper half, the robot moves forward. The lower half is responsible for backward movement. The further the mouse moves away from the origin, the faster the robot moves. The Java-applet sends one packet of information containing the direction and the speed to each motor. The packets are sent in the "MouseMotionListener" method "mouseDragged". That is, packets are only sent when the mouse moves.
To extract the directions and velocities a simple coordinate transformation is made. First, the width and the height of the panel are saved. The panel's coordinate system has its origin in the low left corner. To get the coordinates of the mouse position the functions "getX" and "getY" are used. Then, the coordinates are transformed to a coordinate system in the middle of the panel. This is illustrated in Figure 6.2.

```
public void mouseDragged(MouseEvent e) {
  double height = MouseNavigator.getSize().height;
```

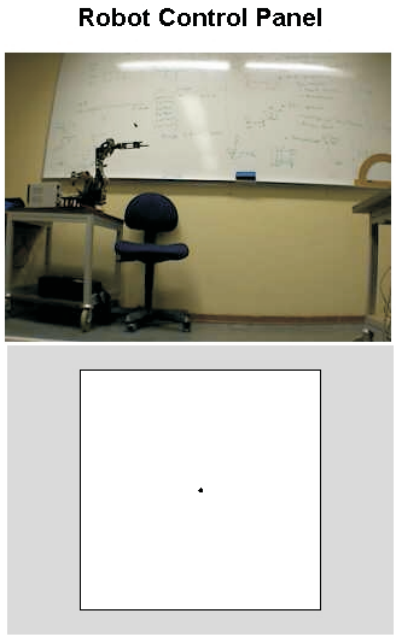

Figure 6.1.: The Java-applets appearance on the robots web site.

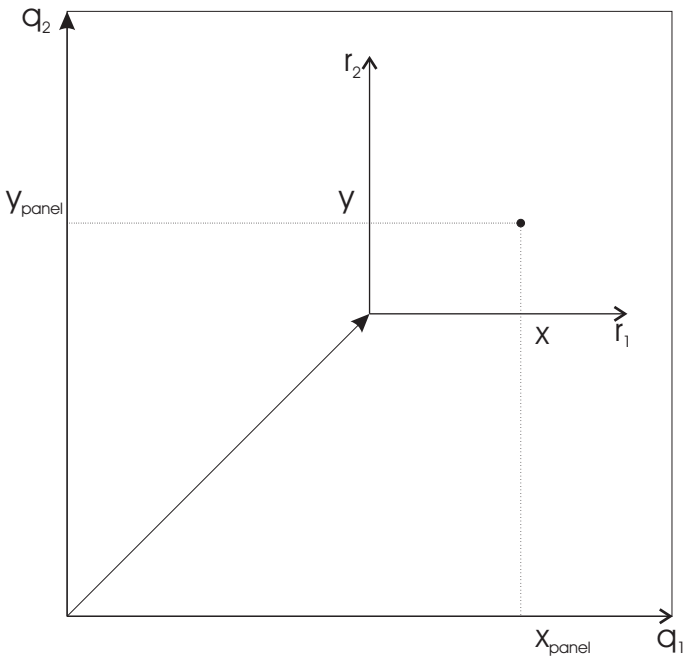

Figure 6.2.: Shift of the coordinate system from the lower left corner to the middel of the panel.

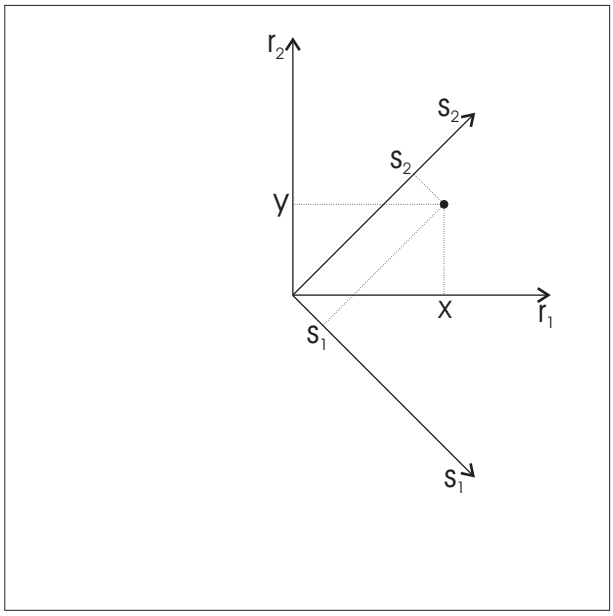

Figure 6.3.: Rotation of the coordinate system and mirroring of the s_2 axis at the s_1 axis.

```
  double width = MouseNavigator.getSize().width;
  double x = (e.getX() - width / 2);
  double y = (e.getY() - height / 2);
...
```

The objective of the coordinate transformation is to store the coordinates directly as a proportional value to the speed. Therefore, the x-coordinate and the y-coordinate must be equal in the absolute value when the mouse points straight ahead. But since the robot's motors are arranged to a differential drive, the motors have to move in different directions to move the robot straight ahead. Thus, a further coordinate transformation is necessary, Figure 6.3. The following code realizes that transformation:

```
...
  double s1 = Math.sqrt(0.5) * x + Math.sqrt(0.5) * y;
  double s2 = Math.sqrt(0.5) * x - Math.sqrt(0.5) * y;
...
```

Now, the coordinates s_1 and s_2 are directly proportional to the motor speeds.
The next problem to solve is that the defined protocol 5.1 of information transfer does not allow to send values of datatype "double". It only allows sending numbers that are

not longer than 20 bit. Furthermore, it is necessary to set the speed between 0x000 and 0x17F. Thus, the obtained "double" values have to be scaled:

```
...
  int IntCast1;
  int IntCast2;
  int sign1=0,sign2=0;
  IntCast1=(int)(s1*7);
  IntCast2=(int)(s2*7);
  if (IntCast1>383)
      IntCast1 =383;
  if (IntCast2>383)
      IntCast2 =383;
  if (IntCast1<-383)
      IntCast1 =-383;
  if (IntCast2<-383)
      IntCast2 =-383;
...
```

First, the s_1-coordinate and the s_2-coordinate are cast to integer numbers and at the same time the values are multiplied by seven. This multiplication is done to get the value of 383 (383 represents the speed maximum) when the mouse points to the highest position in the middle of the panel. Since locations of equal speed form a circle around the panel's origin the values of the casted numbers are too large in the corners of the panel. Therefore, the already casted values have to be set to a maximum value of 383 (0x17F).
Now, the sign has to be treated. If value of the casted numbers "IntCast1" and "IntCast2" are negative, the flag "sign1" or "sign2" is set:

```
...
  if (IntCast1<0) {
      sign1=1;
      IntCast1 =Math.abs(IntCast1);
  }
  if (IntCast2<0) {
      sign2=1;
```

```
      IntCast2 =Math.abs(IntCast2);
  }
...
```

Hence, a capable value to send to the PWM unit is available in the variables "IntCast1" and "IntCast2".
The motor commands can be sent via UDP socket connection from the Java-applet to the Etrax computer. For this, the method "SendMessageViaUdp" is used. The function expects four parameters. The first parameter specifies the command to execute in the receiving device. The second parameter defines the channel to which the data has to be sent. "IntCast1" and "IntCast2" are the values itself and the last parameter specifies the sign of the values:

```
...
  SendMessageViaUdp("1","00",IntCast1, sign1);
  SendMessageViaUdp("1","01",IntCast2, sign2);
}
```

After the reception of the motor commands, the Etrax computer forwards them to the Atmega16 controller. The Atmega16 sends the data packet over the TWI bus to the Atmega8 controller. Hence, the channel and command selection method causes the speed values to be written in the PWM registers OCR1A and OCR1B.

6.2. Results

The sections above describe the components as well as the used programming techniques to set up the mobile robot. All components of the robot work well. Indeed, a problem is the robot speed. It takes too much time to transfer the camera images to the Java-applet. Thus, the frame rate is very slow and if the robot moves too fast a fluent video-stream is impossible to guarantee. Therefore, the robot has to move very slowly so that the user can see all changes in the robot's environment with an acceptable delay. To increase the frame rate, a fast image delivery has to be guaranteed.

7. Autonomous Navigation

Navigation is a quite new area in mobile robotics. Thus, plenty of different approaches have been developed during the last years. The approaches differ concerning the inherited knowledge of the robot.
One approach assumes no knowledge of the environment which the robot is moving in. The robot is given a trajectory that it follows. As a reference, only encoder information of the wheels is used. If obstacles appear, the robot plans its way around them and tries to get back on the trajectory.
Another approach supplies the robot with information about the environment. That is accomplished by using a map in the robot software. This map gives the robot information where unmoveable objects are located. Hence, it is possible to use external information such as locations of fixed objects to orientate in the environment.
A third method commonly used is an approach to build a map during the operation of the robot. This means that the robot does not know anything about its surroundings. During the operation, sensory data is used to build a map gradually. Thus, the robot gets more and more knowledge during the period of operation and accordingly navigation is easier. This method is called "Simultaneous localisation and Map building" - in short: SLAM, [4].
In this thesis a form of the second approach is used. A simple map is available consisting of locations in the corridor the robot is moving in. To navigate, data from motor encoders as well as visual information from the network camera is used. An algorithm detects the lamps on the ceiling. The robot knows where these lamps are located and therefore it knows its own position.
The following section describes the environment of the robot as well as the necessary tasks which have to be done to accomplish smooth navigation.

7.1. The Robots Abilities and the Environment it is Moving in

The robot operates in a corridor with several doors on both sides. A schema of this corridor is shown in Figure 7.1. It is crucial that the robot is able to find a room and turn itself, so that the user can look into the room with the help of the robot's camera. Therefore, the Java-applet which had been described before is extended with some buttons. Through the applet the user can switch to automatic mode which means that the robot takes control of itself. Via the applet, a room can also be chosen to which the robot is supposed to move. The automatic mode can be interrupted by just clicking on the control panel. By doing this, the robot switches to "remote control mode" what means that from now on the user can steer the robot with the mouse. In the current stage of development, the robot has to start at the beginning of the corridor to guarantee correct navigation.

To achieve accurate operation, different problems have to be solved. The largest challenge is that the robot does not drive straight when both motors are provided with the same PWM signal. Hence, controllers for the both motors are necessary. Using a PI-controller for each motor makes the robot drive reasonably straight. The behaviour when moving straight can be further enhanced by adding a third controller, which compensates for the speed difference between the two wheels. However, the encoders which supply the PI-controllers with speed information, cannot measure a difference concerning the wheel diameter. Also, different friction of the wheels will be not noticed. These errors result from the fact that the encoders just measure the turning rate of the wheels and not the real distance which is covered. These errors are integrated over time and result in a drift movement of the robot. Since these errors are too big when moving through the whole corridor, the aid of visual information is necessary to achieve straight driving. Image processing recognizes the lamps on the ceiling. Since these lamps are centred in the corridor, they represent a capable reference for the robot. When the image processing algorithm recognises a lamp, the robot checks whether the lamp is centred in the image

Figure 7.1.: Schema of the corridor - the vertical bars symbolizes the Lamps on the ceiling.

or not. If it is centred, the robot is also centred in the corridor. If the image is not centred, a PID-controller adjusts and turns the robot until the lamp is in the middle of the image. Through this, the integrated error of the motor controllers can be corrected by using the lamps as landmarks.
Furthermore, the robot should not only be able to go straight through the corridor. It should also be possible to head for a special room. In this context a difficulty occurs when a room is found. Then the robot heads towards the door to look in the room. To move further, the robot is supposed to turn back on the corridor. Since the track is lost by turning towards the room, all references are lost as well, so just encoder information can be used to adjust the right heading of the robot and consequently to carry on with the straight movement in the corridor.
The next section, 7.2, presents the basic mathematical theory of the pattern recognition algorithm. In Section 7.3 this mathematical method is used to support the robot in its moving. Section 7.4 exposes how simple navigation can be done by recognizing so called "landmarks" in the robot's surroundings.

7.2. Correlation Coefficient - Mathematical Basics

In this work, pattern recognition is done with the mathematical method of calculating the correlation coefficient. This section explains the mathematical basics and characteristics of this method.
The correlation coefficient indicates the degree of linear dependence of two statistic variables X and Y. All points (X, Y) lie with the probability 1 on a straight line if the correlation coefficient is equal to 1. The variables X and Y are statistically independent if the correlation coefficient equals to 0.
The correlation coefficient is defined by:

$$\varrho_{\mathrm{XY}} = \frac{\sigma_{\mathrm{XY}}}{\sigma_{\mathrm{X}}\sigma_{\mathrm{Y}}} \tag{7.1}$$

X and Y are only independent if $\sigma_{\mathrm{XY}} = 0$. Additional, X as well as Y have to be normally distributed for the independency of X and Y, [8]. Here it is remarked that in case of X and Y being images, this condition is only fulfilled if the image is rich in contrast.
The nominator of Equation 7.1 represents the covariance of X and Y:

$$\sigma_{\mathrm{XY}} = E((X - \mu_{\mathrm{X}})(Y - \mu_{\mathrm{Y}})) \tag{7.2}$$

The covariance also provides a measure for the dependence of X and Y but without standardisation. Only using the covariance leads to a certain problem concerning pattern recognition. If a pattern has to be found within an image that has large areas of white, the covariance will often have a maximum in these regions even if the pattern is contained in the image on another position. This results from the fact that a white pixel has the highest value in an image. And if the highest value is multiplied with any pixel of a pattern, the result will often be a higher value than the value of a multiplication from a pixel with itself. Because of this, it is necessary that the covariance is standardised. This is done by dividing the covariance with the multiplication of the standard deviations of X and Y:

$$\sigma_{\mathrm{X}} = \sqrt{E((X - \mu_{\mathrm{X}})^2)}$$

$$\sigma_{\mathrm{Y}} = \sqrt{E((Y - \mu_{\mathrm{Y}})^2)}$$

The expressions under the square root in the equations above can be estimated in the discrete case when having a finite number k of samples:

$$\sigma_{\mathrm{X}} \approx \sqrt{\sum_{k} (x_{\mathrm{k}} - \mu_{\mathrm{X}})^2 p_{\mathrm{x,k}}}$$

$$\sigma_{\mathrm{Y}} \approx \sqrt{\sum_{k} (y_{\mathrm{k}} - \mu_{\mathrm{Y}})^2 p_{\mathrm{x,k}}}$$

Also, the covariance of Equation 7.2 can be approximated:

$$\sigma_{\mathrm{XY}} \approx \sum_{k} (x_{\mathrm{k}} - \mu_{\mathrm{X}}) p_{\mathrm{x,k}} (y_{\mathrm{k}} - \mu_{\mathrm{Y}}) p_{\mathrm{y,k}}$$

Here, $p_{\mathrm{x,k}}$ is the probability of appearence of the element x_{k} and $p_{\mathrm{y,k}}$ is the probability

of appearence of the element $y_{\mathrm{y,k}}$ respective.
If a static stochastic process is assumed, the standard deviation turns into the auto correlation and the covariance turns into the cross correlation:

Autocorrelation functions:

$$\sum_k (x_\mathrm{k} - \mu_\mathrm{X})^2 p_\mathrm{k} = \frac{1}{N} \sum_k (x_\mathrm{k} - \bar{x})^2$$

$$\sum_k (y_\mathrm{k} - \mu_\mathrm{Y})^2 p_\mathrm{k} = \frac{1}{N} \sum_k (y_\mathrm{k} - \bar{y})^2$$

Crosscorrelation function:

$$\sum_k (x_\mathrm{k} - \mu_\mathrm{X}) p_{\mathrm{x,k}} (y_\mathrm{k} - \mu_\mathrm{Y}) p_{\mathrm{y,k}} = \frac{1}{N^2} \sum_k (x_\mathrm{k} - \bar{x})(y_\mathrm{k} - \bar{y})$$

Since a concrete number of N samples are assumed, the expected value μ_X is exchanged by the average value x_k. Setting these expressions in the original Equation 7.1, leads to:

$$\varrho_{\mathrm{XZ}} \approx r_{\mathrm{xy}} = \frac{\sum_k (x_\mathrm{k} - \bar{x})(y_\mathrm{k} - \bar{y})}{\sqrt{\sum_k (x_\mathrm{k} - \bar{x})^2 \sum_k (y_\mathrm{k} - \bar{y})^2}} \tag{7.3}$$

To compare two images of the same size, r_{xy} can be calculated which gives a measure of how similar two images are.

7.3. Motor Controllers - Straight Movement

Even if two drive motors of the same type are used, they do not have the same speed. Because of tolerances, the two motors differ minimally concerning the motor characteristics including the speed. These minimal differences are enough to let the robot drive along a curve when providing both motors with the same voltage. Since a wheel encoder is available for both motors, it is possible to compensate these differences with a controller. Figure 7.2 shows the control loop for one motor. Since a microcontroller is used to implement the controller, the control system is a sampled data control system. The reference input is formed by the desired angular speed of the motor. Since the

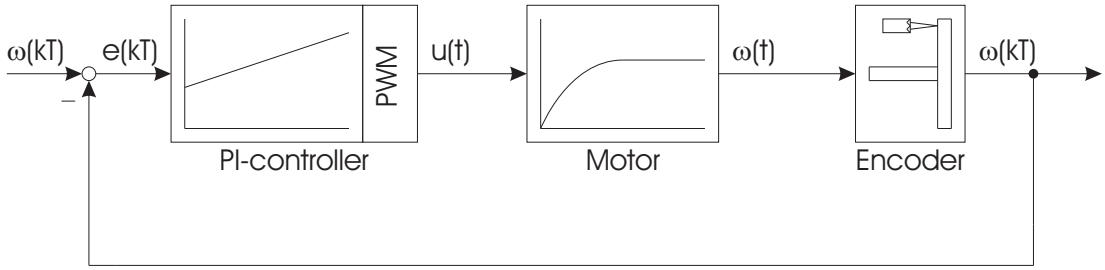

Figure 7.2.: Control loop of one motor

computation power in a microcontroller is restricted a PI-controller is used. The output of the discrete controller is fed to the PWM unit which generates the actuating variable $u(t)$. The motor, a second order lag element, represents the plant of the control loop system and its output is the controlled variable - the motor's speed. The angular speed is measured by the encoder. The output of the encoder is available to the software and the controller algorithm.

7.3.1. Recursive Control Algorithm

The function of a continuous PI-controller is given by the following equation:

$$y(t) = K\left[e(t) + \frac{1}{T_\mathrm{I}}\int e(t)d\mathrm{t}\right]$$

The corresponding discrete control equation is:

$$y_\mathrm{k} = K\left[e_\mathrm{k} + \frac{1}{T_\mathrm{I}}\sum_{\mathrm{i}=1}^{\mathrm{k}} e_\mathrm{i}T\right]$$

In these and the following equations T refers to the sampling time and T_I is the integration constant. The input of the controller is the control error e and the output is represented by y.
To implement a discrete PI-controller, the form of the recursive PI-control algorithm is necessary. To obtain the recursive form, the equation for the control signal at sample time $\mathrm{k}-1$ is subtracted from the equation for y_k:

$$y_{k-1} = K \left[e_{k-1} + \frac{1}{T_I} \sum_{i=1}^{k-1} e_i T \right]$$

$$y_k - y_{k-1} = K \left[e_k - e_{k-1} + \frac{T}{T_I} e_k \right]$$

Bringing y_k to the front, results in the recursive equation:

$$y_k = y_{k-1} + K \left[\left(1 + \frac{T}{T_I} \right) e_k - e_{k-1} \right] \tag{7.4}$$

7.3.2. The Motor Control System

Implementing the above algorithm to control the velocity of each motor gives rise to good results concerning the lateral shift when driving straight. The control accuracy can be further enhanced by adding another control loop which connects the two separate control loops of the motors. This middle control loop corrects the speed difference between the two separate control loops. Hence, if for some reason one motor gets stuck, the other motor's speed will be reduced.
The whole motor control system is shown in Figure 7.3. Since the robot is supposed to move straight, the reference inputs for both motors are equal. To compensate the difference of the motor speeds, an I-controller is used. Its input consists of the measured velocities of both motors.
Added to the input of the I-controller is a "bias" that can be used to add an artificial disturbance that results in driving a curve. Also, this input can be used to compensate differences in the wheel circumference. The recursive, discrete I-controller is given by:

$$y_{i,k} = y_{i,k-1} + \frac{T}{T_i} e_{i,k}$$

7.3.3. Implementation on the Atmega8

To ensure a constant sampling time, the controller is realized in an interrupt routine. As the interrupt source is chosen the overflow interrupt of timer 2. Since the prescaler of timer 2 is chosen as 1024, an interrupt occurs with the frequency of 56,25 Hz. However,

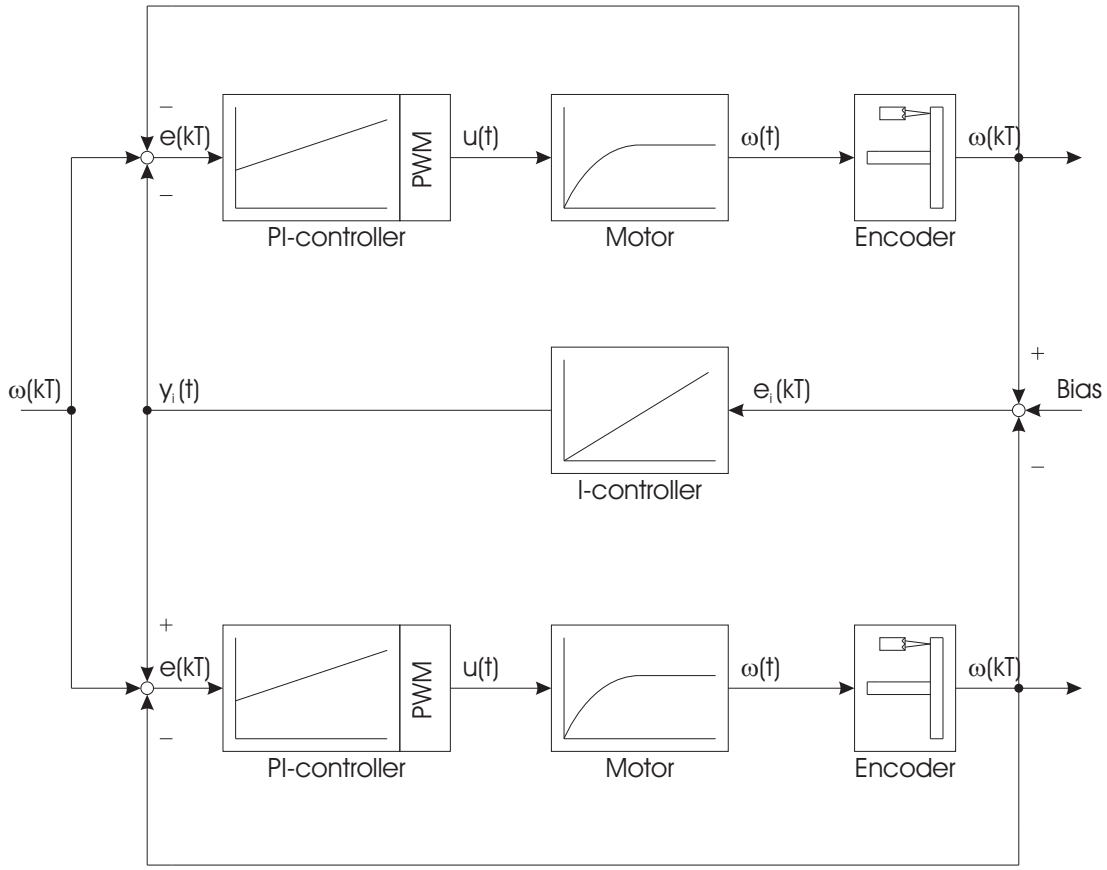

Figure 7.3.: The motor control system realized on the Atmega8 controller.

just every fourth time an interrupt occurs, the control algorithm is executed. Therefore, the control frequency will be 14,06 Hz.
The following code segment shows the implemented interrupt routine. The variables "speed_l" and "speed_r" contain the reference input. These values are converted to encoder ticks. The factor the speed is multiplied with means that every time the control algorithm is executed, 0x29 encoder ticks are counted at a speed of 0x1ff. Here, speed refers to a value in the PWM register. The converted speeds are contained in the variables "des_r" for the right motor and "des" for the left motor:

```
SIGNAL (SIG_OVERFLOW2)
{
  cou++;
  if (cou==4)
    {
      cou=0;
      des_r=speed_r*0x29/0x1ff;
      des=speed_l*0x29/0x1ff;
...
}
```

The encoder ticks counted during one period are contained in the variables "pos1" for the right motor and "pos2" for the left motor respectively. With the help of these variables which form the controlled variables, the control error e_i of the middle control refeeding can be calculated. Having e_i, the output of the middle I-controller y_i can be computed as well:

```
...
      e_i = pos2-pos1+bias;
      y_i= y_i_old + h1*e_i;   // h1=T/Ti
      y_i_old = y_i;
...
```

Here, the auxiliary number $h1$ is brought in. This number represents the I-controller's time constant which is pre computed. All time constants in the control algorithm are pre computed when the variables are initialized. This is done to avoid floating point operations in the interrupt routine.

Next, the control errors of the control loops directly controlling the motor speeds are computed:

```
...
    e_r=des_r-pos1+y_i;
    e=des-pos2-y_i;
...
```

Now, the encoder variables have to be reset and the control signals are computed according to Equation 7.4. $h3$ is also a pre computed constant:

```
...
     pos1=0;
     pos2=0;

     u_r=u_old_r+h3*e_r+e_r_old;  //h3=Kr*(1+T/(Tn*2))
     u=u_old+h3*e+e_old;
...
```

Since the actuating variables as well as the control errors from the last iteration are necessary to compute the actuating variable, they will be saved. To set the motor direction flag the function "SetMotorDirection" is invoked. Also, the speed has to be limited which is done with the function "LimitSpeed":

```
...
     e_old=e;
     e_r_old=e_r;
     u_old_r=u_r;
     u_old=u;

     SetMotorDirection(u_r,u);
     LimitSpeed(u_r,u);
...
```

The last step is to write the actuating values to the PWM registers:

```
...
     u_r1=abs(u_r);
```

```
      u1=abs(u);

      OCR1A=(uint16_t)u_r1;
      OCR1B=(uint16_t)u1;
   }
}
```

7.3.4. Track Correction at Regular Intervals Using Pattern Recognition

The described motor control system works rather well when the distance that the robot has to move is short enough. However, the corridor where the robot is moving in is about twenty meters long so the integrated error is not negligible.
To support the motor control system, image processing can be used. Lamps are situated on the ceiling in the middle of the corridor. The camera of the robot continuously takes images of the corridor. If the robot moves straight, the lamps will be in the middle of the images but if it drives in a curve, the lamps will be shifted to the outer regions of the image. Thus, it is possible to detect when the robot goes off the straight line.
As the robot moves through the corridor the lamps grow larger the nearer they come. When a lamp reaches the upper border of the image, it will be recognized by the correlation algorithm. After the lamp's position is determined, the robot corrects its position by using a PID-controller until the lamp is centred in the middle of the image.
After correcting the position, the robot moves only with the support of the motor control system from section 7.3.3. When another lamp is found, the robot corrects its position again. Due to this control behaviour, the robot moves as shown in Figure 7.4.
In Subsection 7.3.1 the recursive equation for the PI-controller is derived. Deriving the recursive equation for the PID controller leads to the following Equation:

$$y_{\mathrm{k}} = y_{\mathrm{k-1}} + K\left[\left(1+\frac{T_{\mathrm{D}}}{T}\right)e_{\mathrm{k}} - \left(1-\frac{T}{T_{\mathrm{I}}}+2\frac{T_{\mathrm{D}}}{T}\right)e_{\mathrm{k-1}} + \frac{T_{\mathrm{D}}}{T}e_{\mathrm{k-2}}\right] \tag{7.5}$$

This PID-equation is realized in Matlab. The control algorithm always is executed when a lamp is detected but not centred in the middle concerning horizontal displacements.
Next, the most important parts of the PID-controller implemented in Matlab are described.
First, the control error is computed. The number "52" denotes the reference input and

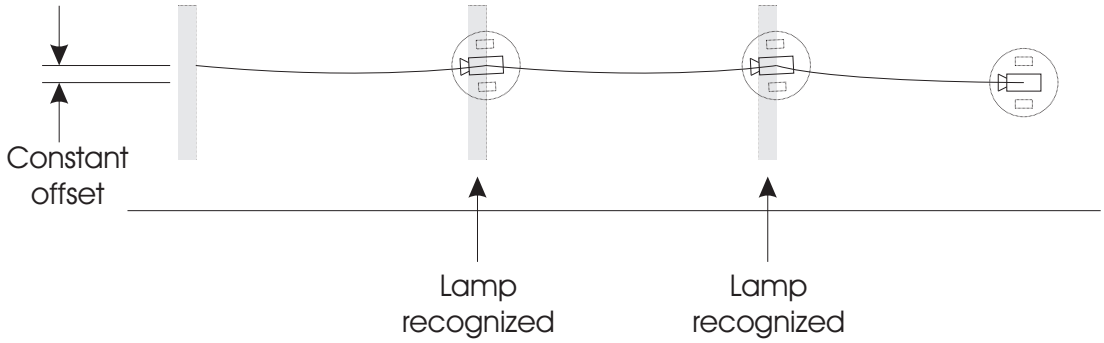

Figure 7.4.: Track of the robot - The vertical bars indicate the recognition of a lamp. It means not that a lamp is located on this position. A recognized lamp is always a couple of meters in front of the robot.

"ind" is the index concerning the column where the lamp is found in the image. Having the control error, the actuating variable can be calculated:

```
...
  e = 52 - ind;
  u = u_old + Kr*(1+Tv/T)*e - Kr*(1-T/Tn+2*Tv/T)*e_1 + Kr*Tv/T*e_2;
...
```

If the actuating variable u is too small because of the static friction the robot won't move. To get an immediate reaction, this dead zone is compensated by assigning a minimum value to the actuating variable:

```
...
  if (e<0)
    if u>=-90
      u=-90;
    end
  else
    if u<90
      u=90;
    end
  end
...
```

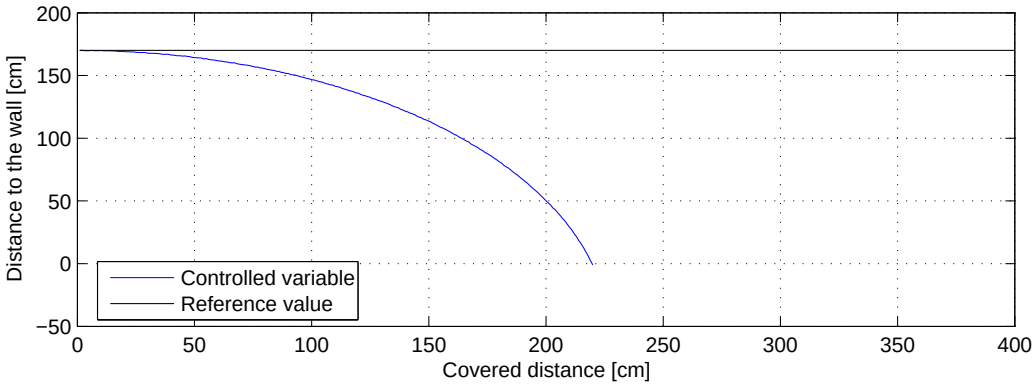

Figure 7.5.: System behaviour without controlling the motor speed.

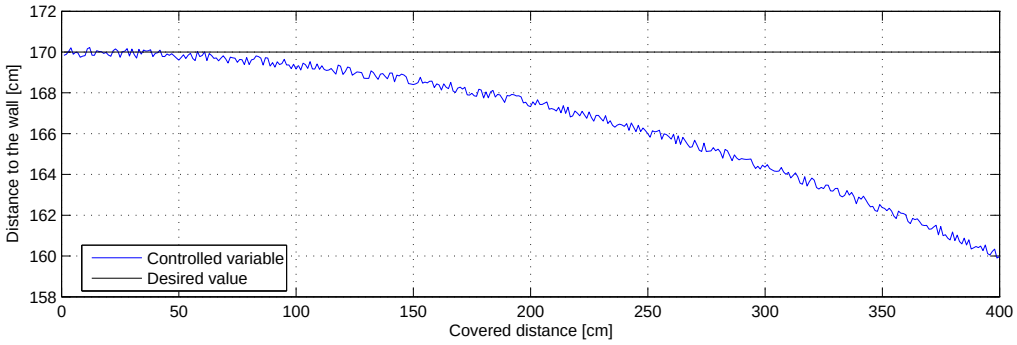

Figure 7.6.: Track of the robot with controlled motor speeds

After this, the old values of the control error and the actuating variable are saved for later use in the next iteration. Then the actuating variable is limited and sent via UDP to the C-program running on the Etrax which forwards it to the Atmega controller board.

7.3.5. Results

The uncontrolled system is compared with the controlled System. To compare both systems, the robot is supposed to drive straight a distance of 4 meters.
First, the uncontrolled system is considered. To drive straight the values $0xbf$ are written to the PWM registers OCR1A and ORC1B. Figure 7.5 shows the covered track. It is aim to drive straight and hence to keep the distance of 170 cm from the wall. It is obvious that both motors have different speeds. The left motor is much faster than the right one and therefore the robot drives in a curve.

Figure 7.6 shows the moved track by using two independent controllers of both motors. The reference variable is set to the same value. It can be seen that the controller action is much better than without controller. The offset at the end of the 4 meter track is just 10 centimeters. Since the robot is adjusted with the help of pattern recognition and the distance between two Lamps is just 2,4 meters, this is an acceptable value.
Using the middle I-controller loop does not lead to better results. The results just would be better if one wheel would be blocked and the controller could not compensate the control error.

7.4. Navigation in the Corridor

As mentioned the robot uses a simple map to navigate in the corridor. When the robot starts moving at the beginning of the corridor it assumes that the first recognized light is the second light in the corridor. The rooms are represented by distances from the lights. For instance, if the user gives an orders to the robot to look in the first door, the robot knows that after recognizing the first door it still has to move $0x3300$ encoder ticks forward to reach the first door.

7.4.1. Recognition of a Lamp

To perform the image processing, Matlab is utilized. The computer on which the image processing is conducted possesses a 350 MHz Pentium processor. Image processing demands a significant amount of computing power, and hence the pattern recognition algorithm can only be applied to very small segments of the images.
The lamps on the ceiling are chosen as reference patterns for several reasons. As mentioned in Section 7.2 the correlation can just be used for pattern recognition if the images are normally distributed. Thus, the pattern which has to be found has to be rich in contrast. This is obviously the case when a lamp with a frame around the lamp is used as a pattern.
Furthermore, a pattern must be located in the middle of the corridor since the location of the pattern is also used to support the motor control system by steering the robot straight in the middle of the corridor.
A third reason to choose the lamps is that the lamps always leave the image at the upper image edge when the robot is driving through the corridor. Thus, the image processing has just to be done in the upper part of the image which saves computing power.
Since the aperture angle of the camera is small, the first lamp seen in the image is the

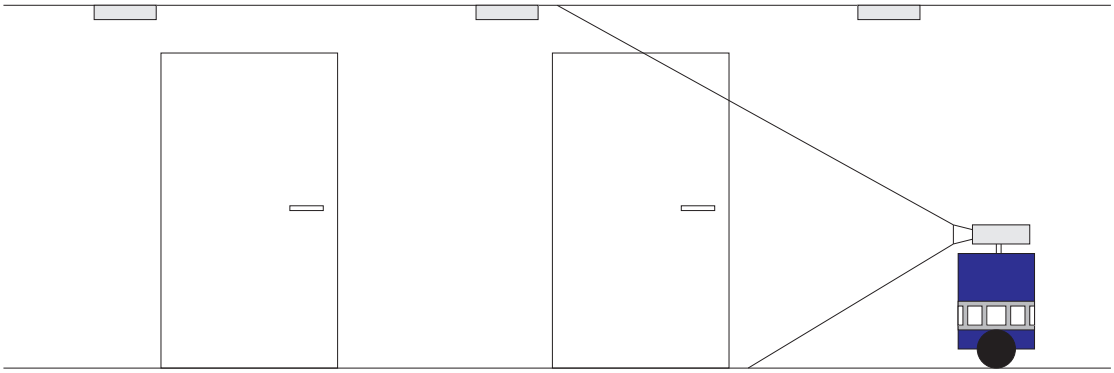

Figure 7.7.: Recognition of a lamp.

second lamp counting the lamps from the robot, see Figure 7.7.
To save additional compute power, only 160×120 images are used instead of the 320×240 images. The pattern which has to be recognized is taken out of the middle of a sample image and has the size of 51×8 pixels.

7.4.2. Windowing the Image and Finding the Pattern

The Equation 7.3 to calculate the correlation coefficient can be applied to two images of the same size. Hence, the image taken from the network camera has to be windowed. This is accomplished by moving a window over the part of the image the pattern has to be searched in. It is started in the left upper corner of the image and after calculating the correlation coefficient, the window is moved one pixel further. Then, the correlation coefficient is computed again and so on. For every position of the window one correlation coefficient is obtained. All correlation coefficients are stored in the correlation matrix. How the correlation coefficients are stored is illustrated in Figure 7.8.

The pattern, in this case a lamp, is found when the maximum of the elements of the correlation matrix is above a pre defined value. The location of the pattern is determined by the indices of the maximum in the correlation matrix.
The following describes the most important parts of the Matlab program responsible for the pattern recognition. The program is executed in a loop. Before the loop starts, some variables are initialized for later use. Also, the pattern which is an image of a lamp is loaded in an array.
Line two of the following code segment extracts just a gray image and since some functions of the Matlab Image Processing Toolbox [9] needs "double" values, the pattern is converted to "double":

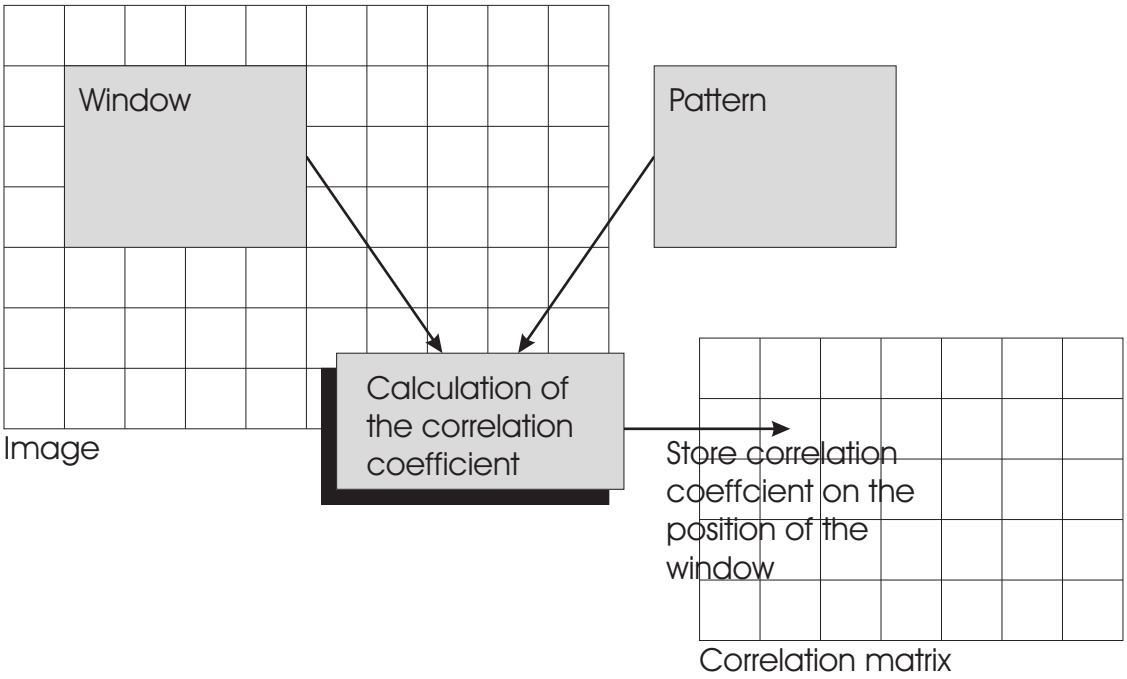

Figure 7.8.: Obtaining the correlation matrix by calculating the correlation coefficient of the pattern and the windowed image.

```
...
pattern=imread('/local/jens/scene6/light2','jpg');
pattern=pattern(:,:,1);
pattern=double(pattern);
...
```

The loop, in which the pattern recognition is done, starts with downloading an image from the network camera:

```
...
while (1)
  I=imread('http://192.168.0.91/axis-cgi/...
              ...jpg/image.cgi?resolution=160x120','jpg');
  I=I(:,:,1);
  I=double(I);
...
```

Now, two "for" loops are started. They move the window through the parts of the image in which the pattern is searched. The matrix "I_w" contains the windowed part of the

image and has the same size as the pattern.
The windowed image as well as the pattern is passed to the function "corrco". Since Matlab interprets the commands, which is slow concerning the execution speed, the calculation of the correlation coefficient is implemented in a "mex" file that contains C-code. This file can be compiled and is much faster than the interpreted commands in Matlab. The function "corrco" calculates the correlation coefficient of the windowed image and the pattern and gives the value back to the correlation coefficient matrix "CorrCoeff":

```
...
[Rows,Cols] = size(pattern);
CorrCoeff = zeros(120-Rows+1,160-Cols+1);
for i=1:3
  for j=10:160-Cols-10
    I_w = I(i:i+Rows-1,j:j+Cols-1);
    CorrCoeff(i,j) = corrco(I_w,pattern);
      if CorrCoeff(i,j)>0.94
        if (CorrCoeff(i,j)>maxim)
          maxim=CorrCoeff(i,j);
          ind=j;
        end
      end
  end
end
...
```

Some experiments showed that a correlation coefficient of 0,94 is a reliable sign that a lamp is found. Therefore, only values above 0,94 are saved and at the same time the maximum of the correlation matrix is searched.
After a lamp is found the maximum will be higher than 0,94 and thus the "if" condition in the following code segment is fulfilled. If now the pattern's position is between pixel 50 and 54, the robot is assumed to be centred. The variable "c2" is increased when a pattern is found more than one times in a row. And if a pattern is found 4 times in a row between the pixel 50 and 54, the lamp and so the robot is assumed to be stable in the center. Then, the message "<light>>", which indicates that a light is found, can be sent via the UDP connection to the Etrax computer:

```
...
if (maxim>0.94)
  if ((ind>=50) & (ind<=54))
    c2=c2+1;
    if c2>=4
      c2=0;
      dat='<light>>';
      try, % Failsafe
        pnet(udp,'write',dat);
        pnet(udp,'writepacket',host,port);   % Send buffer as UDP packet
      end
    end
...
```

How the Etrax computer deals with this message, is described in the following subsection. If the light is found but not centered in the image, the PID-controller described in Section 7.3.4 compensates the offset.
Remark: The "mex" file which implements the function "corrco" is described in the Attachment A.

7.4.3. Simple Navigation - Counting the Lamps

If a lamp is properly recognized, the message "<light>>" will be sent via a UDP socket connection to the Etrax computer. The C-program on the Etrax computer distinguishes between mainly two kinds of messages. A message starting with the character "<", indicates that a part of the data processing has to be done on the Etrax computer. If not such a character is the first in the message, the message is directly forwarded to the Atmega controller.
Since the message contains the string "<light>>", the following code segment is executed:

```
...
  else if (strcmp(msg_r,"<light>>")==0)
    {
      light=light+1;
      if (NoRoom!=light)
        {
          RS232_send("107000ff");
```

```
          waitAmom(400);
          sendUDP_2("<<goon>>");
        }
    }
...
```

The variable "light" is increased. Because of the value of "light", the Etrax computer knows where the robot is located in the corridor. Another variable "NoRoom" denotes the room in which the robot is supposed to look in. If the room number and the number of lights are not equal, the robot will move further and with the command "sendUDP_2" Matlab is directed to go on with finding lights.
In case the variables "NoRoom" and "light" are equal, the robot knows that the next door is the addressed door at which the robot has to stop. Therefore, no drive command is sent to the Atmega controller. In this case another thread of the C-program on the Etrax computer takes control.
The thread "GoToRoom" waits in the background until the variables "NoRoom" and "light" are equal. For instance if the first room is addressed and the two variables are equal, the first "case" of the switch structure in the following code segment is executed. First, a command is sent to the Atmega to drive straight with the velocity $0xff$. The second message sent via the serial port to the Atmega controller is a command on which the Atmga controller reacts by sending the up to date value of the encoder. This value of the encoder is received by the Etrax computer and saved to the variable "position". Now, the Etrax asks for new encoder data as long as the value is smaller than $0x3300$ because $0x3300$ encoder ticks have to be waited until the robot moves from the position where the light is found to the door. After reaching this value, the robot stops:

```
...
void *GoToRoom(int *ii) {
  while (1) {
    switch (NoRoom)
      {
      case 0: break;
      case 1:
      {
      if (light==1)
        {
```

```
          RS232_send("107000ff");
          RS232_send("10400000");
          while (position<0x3300)
            { waitAmom(10); RS232_send("00400000"); }
          waitAmom(10);
          RS232_send("10700000");
...
```

In order to look into the room, the robot has to turn towards the door. This is done by driving the left motor with the speed $0xAf$ while keeping the right motor turned off. Now, constantly encoder information is asked until the value of the encoder reaches $0x750$. Then, the robot looks direct in direction of the room:

```
...
          // turn towards the door
          RS232_send("10400000");
          RS232_send("101000af");
          position=0;
          while (position<0x750)
            { waitAmom(5); RS232_send("00400000"); }
          waitAmom(5);
          RS232_send("10700000");
          NoRoom=0;
        }
      break;
    }
...
```

Remark: The function "waitAmom" contains "for" loops to wait some time until the next action is done.

8. Conclusion and Outlook

In this thesis, the assembly of a mobile robot is described. In the first part, the single components are explained and how they work together. The robot's hardware consists mainly of the chassis of an old robot, two Atmega microcontrollers, an Etrax computer, a network camera as well as an external personal computer.
The Atmega controller boards form an interface to all hardware devices. The Atmega8 controller provides an interface to the motors. The motors are driven by using the PWM unit of the controller. Also, data from the wheel encoders is read. The data provides the programs, running on the different systems of the robot, with crucial information about the robots velocity and covered distances. The Atmega16 controller represents an interface to read sensory data from the bumper switches. However, even if the physical connection between the controller board and the switches exists, these bumper switches were not used in this project.
Since a central part of this project concerns interfaces, the Etrax computer is a crucial part of the robot. All sent data is received by the Etrax computer and distributed to the different components of the robot. Also, it is used to do some higher level controlling of the robot.
The mobile components are connected over wireless LAN to a host computer where calculations demanding higher computing power are accomplished. Mainly, the pattern recognition which is used for navigational reasons is done on this host. Therefore, the network camera mounted on the robot provides the pattern recognition algorithm with images.
One part of this thesis discusses the used interfaces and their functions in the robot system. The Atmega controllers communicate by using the TWI serial bus. The Atmega and the Etrax computer uses the RS232 serial port to exchange information. All communication to the external host as well as to the Java-applet is done with a UDP socket connection.
After all basic software implementations are made, a remote control of the robot is realized. To control the robot via internet, a user can download an html web page from the

web server located on the Etrax computer. Thus, the user can see an image taken by the network camera on the robot in his browser. Also, an Java-applet is started which provides an interface to steer the robot via the computer mouse.
The last part of this project deals with the implementation of an autonomous function. The objective is that the robot navigates in a known environment and finds rooms on its own. In this context, it is a challenge to guarantee that the robot drives straight while operating autonomous. To drive straight, it is not sufficient to provide the motors with equal voltages. Since the motors are not completely identical in construction, they will have different speeds. To compensate this speed difference, a motor control system consisting of PI-controllers is designed.
To support the motor control system, pattern recognition is used. Lamps on the ceiling which are used as landmarks provide the robot with reference points to orientate in the corridor. These landmarks are also used to accomplish simple navigation. By counting the landmarks, the robot knows where it is located in the corridor and thus it can find a room itself.

The robot system can be extended in further projects by improving its capability to orientate. This can be done for instance by using an artificial landmark at the end of the corridor. Then, the size has to be estimated by an algorithm and so a continuous estimation of the robot's position is possible. However, to implement such an algorithm more computing power is necessary to do the more computationally demanding calculations.
Further, it can be examined if other pattern recognition methods lead to better results. For example, modern approaches like Neural Networks and Vector Support Machines provide reliable classification properties.
Another feature that could be added is the ability to plan the robots movement with trajectories. Thus, the robot could drive along curves just by commanding to drive a special radius with a certain velocity.
Since a robot arm is available, further projects could involve assembling this arm to the robot.

9. Acknowledgement

I want to thank all people who helped me in any way to complete this thesis. I especially want to thank my supervisors Anders Blomdell and Karl-Erik Årzén. During the project, Anders supported me with technical help and always had good suggestions how to solve problems. Karl-Erik made it possible for me to come to Lund and write my thesis here. He helped me concerning organizational issues and he also read my thesis and corrected it. Furthermore, I want to thank Richard Andersson who helped me with some English expressions. I enjoyed the time when we were sitting together by a cup of cappuccino discussing technical terms.
Last but not least, I give thanks to Michael Leykauf and Rolf Isermann who helped me from my home university in Darmstadt to manage organizational things concerning the exchange program.

Bibliography

[1] **Axis Communications AB,** Axis Developer Documentation (ETRAX 100LX), `http://developer.axis.com/doc/index.html`

[2] **General Robotics Corporation,** http://www.generalrobotics.com

[3] **Axis Communications AB,** Linux for the Etrax computer, `http://developer.axis.com/software/linux/index.html`

[4] **M. W. M. Gamini Dissanayake, Paul Newman, Steven Clark, Hugh F. Durrant-Whyte, M. Csorba,** A Solution to the Simultaneous Localization and Map Building (SLAM) Problem

[5] **Årzén Karl-Erik,** Real-Time Control Systems, lecture notes, Department of Automatic Control, LTH 2003

[6] **Atmel,** Atmega8 manual, 2004

[7] **Atmel,** Atmega16 manual, 2004

[8] **Bronstein, Semendjajew, Musiol, Mühlig,** Taschenbuch der Mathematik, 4th revised edition, Verlag Harri Deutsch, 1999

[9] **MathWorks,** Image Processing Toolbox, http://www.mathworks.com/products/image/

[10] **Lutz, Wendt** Taschenbuch der Regelungstechnik, 5th revised edition, Verlag Harri Deutsch, 2003

[11] **B.V.K. Vijaya Kumar, Abhijit Mahalanobis, Richard D. Juday** Correlation Pattern Recognition, Cambridge University Press, 2005

A. The Matlab Function "corrco"

The "mex" function is responsible for the most computing power consuming calculations. Both auto correlation functions as well as the cross correlation of the windowed image and the pattern are computed in the "mex" file.
The "mex" function has a determined number of arguments in its gateway function "mexFunction". The number of left hand arguments is contained in the integer variable "nlhs" and the number of right hand side arguments is contained in "nrhs". The pointer "plhs" is a pointer to an array that contains all input data passed to the "mex" function. The data which is returned can be accessed with the pointer "prhs":

```
#include "mex.h"
#include "math.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, ...
                 ... const mxArray *prhs[])
{
...
```

Now, the declarations of all used variables have to be made. Then, it is checked if the right number of arguments are passed to the function:

```
...
  double *I, *P, *y, acf_p, acf_i, ccf;
  int i, rows_i, cols_i, rows_p, cols_p;
  /* Check for proper number of arguments. */
  if (nrhs != 2) {
    mexErrMsgTxt("Two inputs required."); }
...
```

In the following, for later use, the sizes of the input arrays are saved. In a "mex" file, all arrays only can be accessed by pointes. Therefore, the pointers to the matrices are assigned. Also, the output matrix which is a scalar has to be created:

```
...
  /* The input must be a noncomplex scalar double.*/
  rows_i = mxGetM(prhs[0]);
  cols_i = mxGetN(prhs[0]);
  rows_p = mxGetM(prhs[1]);
  cols_p = mxGetN(prhs[1]);

  /* Create matrix for the return argument. */
  plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);

  /* Assign pointers to each input and output. */
  I = mxGetPr(prhs[0]);
  P = mxGetPr(prhs[1]);
  y = mxGetPr(plhs[0]);
...
```

The last part of the "mex" function is the actual implementation for calculating the correlation coefficient. The correlation coefficient calculated in accordance to equation 7.3 is contained in "y[0]":

```
...
  ccf=0;
  acf_p=0;
  acf_i=0;
  for (i=0;i<rows_p*cols_p;i++) {
    acf_p = acf_p + P[i]*P[i];
    acf_i = acf_i + I[i]*I[i];
    ccf = ccf + P[i]*I[i];
  }
  y[0] = ccf/sqrt(acf_p*acf_i);
}
```

The scalar which is returned to Matlab is "y[0]".

B. Source Code of the Atmega8 Program

```
//#include <avr/io.h>
#include <string.h>
#include <avr/ina90.h>
//#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

// speed control
volatile int speed_r=0;
volatile int speed_l=0;
volatile int bias=0;
volatile int straight=0;

// TWI
volatile int flagT=0;
volatile int flagR=0;
volatile unsigned char dataT[4];
volatile unsigned char dataR[4];

#define MESSAGE_LENGTH 8

volatile unsigned char encoder1, encoder2;
volatile int pos1=0;
volatile int pos2=0;
volatile long pos1sum=0;
volatile long pos2sum=0;

char* makeString(char *cmd, char *channel, uint32_t value);
void motorControlresRight();
void motorControlresLeft();
void Ch7selectCmdAndSendData(uint8_t cmd, uint8_t sign, uint16_t value);

// INITIALISATION-ROUTINES----------------------------------------------------------------
void PORT_init()
{
  PORTB = 0x39; // Port B, pull up PB0, PB3, PB4 & PB5
  DDRB = 0x06;  // PortB, PB1 & PB2 outputs

  PORTD = 0x00; // no pull up
  DDRD = 0xc0;  // PortD, PD6 & PD7 outputs

  PORTC = 0x00; // no pull up
  DDRC = 0x0c;  // PortC, PC2 & PC3 outputs
}

void USART_Init()
{
  _CLI();

  // Set baud rate
  UBRRH = 0x00;
  UBRRL = 23;
  // Enable receiver and transmitter
  UCSRB = (1<<RXCIE)|(1<<RXEN)|(1<<TXEN);
  // Set frame format: 8data, 1stop bit
  UCSRC = (1<<URSEL)|(3<<UCSZ0);
```

```
  _SEI();
}

void TWI_init() {
  _CLI();
  TWAR = 0x44; // own slave addr
  TWBR = 0x0a;
  TWSR = (0<<TWPS1) | (0<<TWPS0);
  TWCR = (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
  _SEI();
}

void PWM_Init()
{
  _CLI();
  ICR1=0x04FF;
  TCCR1A = 0xa0;
  TCCR1B = 0x11;
  TIMSK = 0x00;//04    //interrupt 0x04 - timer1 only
  TCCR2= (1<<CS22) | (1<<CS21) | (1<<CS20);
  OCR1A=0x0;
  OCR1B=0x0;
  _SEI();
}
// -----------------------------------------------------------------------------

void USART_putc( unsigned char data )
{
  // Wait for empty transmit buffer
  while ( !( UCSRA & (1<<UDRE)) ) ;
  // Put data into buffer, sends the data
  UDR = data;
}

void USART_puts(char *data, int lenth) {
  int count;
  for (count = 0; count < lenth; count++)
    USART_putc(*(data+count));
}

uint32_t convertStringToHex(char *string) {
  uint32_t i=0,fourbytes=0,value=0;
  for (i=0;i<MESSAGE_LENGTH;i++)
    {
      if ('0' <= *(string+i) && *(string+i) <='9') {
    value = *(string+i) - '0';
      }
      else if (('A' <= *(string+i) && *(string+i) <= 'F')) {
    value = *(string+i) - 'A' + 10;
      }
      else if (('a' <= *(string+i) && *(string+i) <= 'f')) {
          value = *(string+i) - 'a' + 10;
      }
      fourbytes = fourbytes << 4;
      fourbytes |= value;
    }
return fourbytes;
}

char* makeString(char *cmd, char *channel, uint32_t value)
{
  uint8_t i;
  uint32_t mask=0x000f0000;
  static char sendData[8];
  char hex[5];

  for (i=0;i<5;i++) {
```

```
    int v = (value & mask)>>(16-(i*4));
    if (0 <= v && v <= 9) {
      hex[i] = '0' + v;
    } else {
      hex[i] = 'A' + v - 10;
    }
    mask = mask >> 4;
  }

  sendData[0]=*(cmd+0);
  sendData[1]=*(channel+0);
  sendData[2]=*(channel+1);
  sendData[3]=*(hex+0);
  sendData[4]=*(hex+1);
  sendData[5]=*(hex+2);
  sendData[6]=*(hex+3);
  sendData[7]=*(hex+4);

  return sendData;
}


// Channelhandling for channels on this AVR ------------------------------------------------
void Ch0selectCmdAndSendData(uint8_t cmd, uint8_t sign, uint16_t value) {
  switch (cmd)
    {
    case 0: { break; } // GetValue
    case 1:
      {
    straight=0;
    speed_r=0;
    speed_l=0;
    pos1=0;
    pos2=0;
    OCR1A = value;
    if (sign==0)
      PORTC = PORTC & 0xfb;
    else
      PORTC = PORTC | 0x04;
    break; // SetValue
      }
    case 2: { break;} // further commands
    }
}

// left motor - PB2, dir: PC3
void Ch1selectCmdAndSendData(uint8_t cmd, uint8_t sign, uint16_t value) {
  switch (cmd)
    {
    case 0: { break; } // GetValue
    case 1:
      {
    speed_r=0;
    pos1=0;
    pos2=0;
    speed_l=0;
    straight=0;
    OCR1B = value;
    if (sign==0)
      PORTC = PORTC & 0xf7;
    else
      PORTC = PORTC | 0x08;
    break; // SetValue
      }
    case 2: { break;} // further commands
    }
}
```

```
/*
// Bumper - currently PORTD switches
void Ch2selectCmdAndSendData(long ReceivedRS232Hex) {
  switch ((ReceivedRS232Hex & 0xf0000000)>>28)
    {
    case 0: break; // GetValue
    }
}



//right encoder
void Ch3selectCmdAndSendData(long hex) {
  switch ((hex & 0xf0000000)>>28)
    {
    case 0: break;  // GetValue
    }
}
*/
//left encoder
void Ch4selectCmdAndSendData(uint8_t cmd, uint8_t sign, uint16_t value) {
  switch (cmd)
    {
    case 0: USART_puts(makeString("0","04",pos2sum),8); break;  // GetValue
    case 1:
      {
    pos2sum=0;
    /*    while (pos2sum<value+10) {}
    Ch7selectCmdAndSendData(1,0,0);
    USART_puts(makeString("4","44",pos1sum),8);
    pos1sum=0;*/
    break;
      }
    }
}

//right
void Ch5selectCmdAndSendData(uint8_t cmd, uint8_t sign, uint16_t value) {
  switch (cmd)
    {
    case 1:
      {
    straight=0;
    pos1=0;
    pos2=0;
    TIMSK = TIMSK & 0xbf;
    if (sign)
      speed_r = value;
    else
       speed_r = -value;
    motorControlresRight(); motorControlresLeft();
    TIMSK = TIMSK | 0x40;
    break;
    }
    }
}

//left
void Ch6selectCmdAndSendData(uint8_t cmd, uint8_t sign, uint16_t value) {
  switch (cmd)
    {
    case 1:
      {
    straight=0;
    pos2=0;
    pos1=0;
```

```
    TIMSK = TIMSK & 0xbf;
    if (sign)
      speed_l = value;
    else
       speed_l = -value;
    motorControlresLeft(); motorControlresRight();
    TIMSK = TIMSK | 0x40;
    break;
    }
    }
}

// straight
void Ch7selectCmdAndSendData(uint8_t cmd, uint8_t sign, uint16_t value) {

  switch (cmd)
    {
    case 1:
      {
    pos1=0;
    pos2=0;
    TIMSK = TIMSK & 0xbf;

    if (sign == 0) {
      speed_l = value;//(hex & 0x00000fff);
      speed_r = value;//(hex & 0x00000fff);
    }
    else {
      speed_l = -value;
      speed_r = -value;//-(hex & 0x00000fff);
    }
    straight=1;
    if (value == 0) {
      straight = 0; // middel control off, otherwise ueberschwingen
      speed_l = 0;
      speed_r = 0;
    }
    //bias = (hex & 0x000ff000);
    motorControlresLeft();
    motorControlresRight();
    TIMSK = TIMSK | 0x40;
    break;
    }
    }
}
// ----------------------------------------------------------------------------------

volatile uint32_t hex=0;
uint8_t j=0,chan=0,sign=0,cmd=0;
uint16_t value=0;
char ch[MESSAGE_LENGTH];
SIGNAL (SIG_UART_RECV)
{
  ch[j]=UDR;
  j++;

  if (j==8) {
    j=0;
    hex=convertStringToHex(ch);
    //memset(&ch, 0, sizeof(ch));

    cmd = (hex & 0xf0000000)>>28;
    chan = (hex & 0x0ff00000)>>20;
    sign = hex & 0x000f0000;
    value = hex & 0x0000ffff;

    switch (chan)
```

```
      {
      case 0: TIMSK = TIMSK & 0xbf; Ch0selectCmdAndSendData(cmd,sign,value); break; // this AVR - right motor
      case 1: TIMSK = TIMSK & 0xbf; Ch1selectCmdAndSendData(cmd,sign,value); break; // this AVR - left motor
      case 2: //Ch2selectCmdAndSendData(hex); break; // this AVR - free
      case 3: //Ch3selectCmdAndSendData(hex); break; // this AVR - right encoder
      case 4: Ch4selectCmdAndSendData(cmd,sign,value); break; // this AVR - left encoder
      case 5: Ch5selectCmdAndSendData(cmd,sign,value); break;
      case 6: Ch6selectCmdAndSendData(cmd,sign,value); break;
      case 7: Ch7selectCmdAndSendData(cmd,sign,value); break; // straight=1
      case 8: break;
      }
  }
}

int cou=0;

volatile int e=0;
volatile int e_old=0;
volatile int u=0;
volatile int u_c=0;
volatile int u_old=0;
volatile int des=0;

volatile int e_r=0;
volatile int e_r_old=0;
volatile int u_r=0;
volatile int u_r_c=0;
volatile int u_old_r=0;
volatile int des_r=0;

volatile int ee=0;
volatile int e_i=0;
volatile int e_i_old=0;

volatile int u1=0;
volatile int u_r1=0;
void motorControlresLeft() {
  e=u=0;
  u_c=0;
  u_old=0;
  ee=0;
  e_i=0;
  e_i_old=0;
}

void motorControlresRight() {
  e_r=0;
  u_r=0;
  u_r_c=0;
  u_old_r=0;
  ee=0;
  e_i=0;
  e_i_old=0;
}

int h1=0.0711/0.1;         // T/(2*Ti)
int h3=1*(1+0.0711/0.2); // Kr*(1+T/(Tn*2)
SIGNAL (SIG_OVERFLOW2)
{
  cou++;
  if (cou==4)
    {
      cou=0;
      des_r=speed_r*0x29/0x1ff;
      des=speed_l*0x29/0x1ff;
      ee = pos2-pos1+bias;
      e_i= e_i_old + h1*ee;//T/(2*Ti)*ee;
```

```
      e_i_old = e_i;

      if (straight==1) {
    e_r=des_r-pos1+e_i; //USART_puts(makeString("0","00",(uint16_t)pos1),8);
    e=des-pos2-e_i;
      }
      else {
    e_r=des_r-pos1;//+(-u_r+u_r_c)*0.7; //USART_puts(makeString("0","00",(uint16_t)pos1),8);
    e=des-pos2;//+(-u+u_c)*0.7;
      }

      pos1=0;
      pos2=0;

      e_old=e;
      e_r_old=e_r;

      u_r=u_old_r+h3*e_r+e_r_old;  //Kr*(1+T/(Tn*2))*e_r;  //PI controller
      u=u_old+h3*e+e_old;  //Kr*(1+T/(Tn*2))*e;  //PI controller

      u_old_r=u_r;
      u_old=u;

      if (u_r<0)
    PORTC = PORTC & 0xfb;
      else
    PORTC = PORTC | 0x04;
      if (u<0)
    PORTC = PORTC | 0x08;
      else
    PORTC = PORTC & 0xf7;

      if (u_r>0x17f)
    u_r=0x17f;
      if (u_r<-0x17f)
          u_r=-0x17f;
      if (u>0x17f)
    u=0x17f;
      if (u<-0x17f)
          u=-0x17f;

      u_r1=abs(u_r);
      u1=abs(u);

      OCR1A=(uint16_t)u_r1;
      OCR1B=(uint16_t)u1;
    }
}

SIGNAL (SIG_OVERFLOW1)
{
  // 14.7456MHz/1/1024 -> 14400Hz
  // This should be often enough to get the encoders right
  unsigned char portd, e1, e2;

  portd = PIND;
  e1 = portd & 0x14;
  e2 = portd & 0x28;
  switch ((encoder1) | (e1 >> 1)) {
    case 0x0:
    case 0x3:
    case 0xc:
    case 0xf: {
      // No change
    } break;
    case 0x1:
    case 0x7:
```

```
    case 0xe:
    case 0x8: {
      pos1--;
      pos1sum--;
    } break;
    case 0x4:
    case 0xd:
    case 0xb:
    case 0x2: {
      pos1++;
      pos1sum++;
    } break;
    default: {
      // error = 1;
    }
  }
  switch ((encoder2) | (e2 >> 2)) {
    case 0x0:
    case 0x3:
    case 0xc:
    case 0xf: {
      // No change
    } break;
    case 0x1:
    case 0x7:
    case 0xe:
    case 0x8: {
      pos2++;
      pos2sum++;
    } break;
    case 0x4:
    case 0xd:
    case 0xb:
    case 0x2: {
      pos2--;
      pos2sum--;
    } break;
    default: {
    }
  }
  encoder1 = e1 >> 2;
  encoder2 = e2 >> 3;
}

SIGNAL(SIG_2WIRE_SERIAL) {

  switch (TWSR & 0xf8)
    {
    case 0x60:
      {
    flagR=0;
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    break;
      }
    case 0x80:
      {
    dataR[flagR] = TWDR;
    flagR++;
    if (flagR<3) {
      TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    } else {
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    }
    break;
      }
    case 0x88:
      {
```

```
    dataR[flagR]=TWDR;

    //    uint32_t hex=0;
    int i=0;
    for (i = 0 ; i < 4  ; i++) {
      hex = hex << 8;
      hex |= dataR[i];
    }

    cmd = (hex & 0xf0000000)>>28;
    chan = (hex & 0x0ff00000)>>20;
    sign = (hex & 0x000f0000)>>16;
    value = hex & 0x0000ffff;

    switch (chan)
      {
      case 0: TIMSK = TIMSK & 0xbf; Ch0selectCmdAndSendData(cmd,sign,value); break; // this AVR - right motor
      case 1: TIMSK = TIMSK & 0xbf; Ch1selectCmdAndSendData(cmd,sign,value); break; // this AVR - left motor
      case 2: //Ch2selectCmdAndSendData(hex); break; // this AVR - free
      case 3: //Ch3selectCmdAndSendData(hex); break; // this AVR - right encoder
      case 4: //Ch4selectCmdAndSendData(cmd,sign,value); break; // this AVR - left encoder
      case 5: Ch5selectCmdAndSendData(cmd,sign,value); break;
      case 6: Ch6selectCmdAndSendData(cmd,sign,value); break;
      case 7: {Ch7selectCmdAndSendData(cmd,sign,value); } break; // straight=1
      case 8: break;
      }

    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE); // Stop
      } break;

    // ------------ Slave transmit mode -------------------------------
    case 0xA8:
      {
    flagT=0;
    TWDR=dataT[flagT];
    flagT++;
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    break;
      }
    case 0xB8:
      {
    TWDR=dataT[flagT];
    flagT++;
    if (flagT<4) {
      TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    } else {
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    }
    break;
      }
    case 0xC0:
      {
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    break;
      }
    }
}

int main()
{
  PORT_init();
  USART_Init();
  PWM_Init();
  TWI_init();
  while (1) {
  }
}
```

C. Source Code of the Atmega16 Program

```
#include <stdio.h>
//#include <stdlib.h>
#include <avr/io.h>
#include <avr/signal.h>
#include <string.h>
#include <avr/ina90.h>
#include <inttypes.h>
//#include <avr/interrupt.h>

#define MESSAGE_LENGTH 8

volatile unsigned char read=0;
volatile unsigned char flagR=0;
volatile unsigned char flagT=0;
volatile unsigned char dataR[4];
volatile unsigned char dataT[4];

volatile long pos1, pos2;

void TWI_transmit_4Byte(uint32_t hex);
char* makeString(char *cmd, char *channel, uint32_t value);

// INITIALISATION-ROUTINES---------------------------------------------------------------
void PORT_init()
{
  DDRB = 0xff;  // PortB, PB1 & PB2 outputs
  PORTB=0xff;
}

void USART_Init()
{
  _CLI();

  // Set baud rate
  UBRRH = 0x00;
  UBRRL = 23;
  // Enable receiver and transmitter
  UCSRB = (1<<RXCIE)|(1<<RXEN)|(1<<TXEN);
  // Set frame format: 8data, 1stop bit
  UCSRC = (1<<URSEL)|(3<<UCSZ0);

  _SEI();
}

void TWI_Init() { // this AVR is master
  DDRB = 0xff;
  PORTB=0xff;

  _CLI();
  TWBR = 0x0a;
  TWSR = (0<<TWPS1) | (0<<TWPS0);
  _SEI();
}

// -------------------------------------------------------------------------------

void USART_putc( unsigned char data )
{
  // Wait for empty transmit buffer
```

```
  while ( !( UCSRA & (1<<UDRE)) ) ;
  // Put data into buffer, sends the data
  UDR = data;
}

void USART_puts(char *data, int lenth) {
  int count;
  for (count = 0; count < lenth; count++)
    USART_putc(*(data+count));
}

uint32_t convertStringToHex(char *string) {
  uint32_t i=0,fourbytes=0,value=0;
  for (i=0;i<MESSAGE_LENGTH;i++)
    {
      if ('0' <= *(string+i) && *(string+i) <='9') {
    value = *(string+i) - '0';
      }
      else if (('A' <= *(string+i) && *(string+i) <= 'F')) {
    value = *(string+i) - 'A' + 10;
      }
      else if (('a' <= *(string+i) && *(string+i) <= 'f')) {
          value = *(string+i) - 'a' + 10;
      }
      fourbytes = fourbytes << 4;
      fourbytes |= value;
    }
return fourbytes;
}

char* makeString(char *cmd, char *channel, uint32_t value)
{
  static char sendData[8];
  char hex[8];

  sprintf(hex,"%lx",value);

  strcpy(sendData,cmd);
  strcat(sendData,channel);

  if (value < 17)
    { strcat(sendData,"0000"); }
  if (value > 16 && value < 256)
    { strcat(sendData,"000"); }
  if (value > 255 && value < 4096)
    strcat(sendData,"00");
  if (value > 4095)
    { strcat(sendData,"0"); }

  strcat(sendData,hex);

  return sendData;
}

// ------------------------------------------------------------------------------
SIGNAL(SIG_2WIRE_SERIAL){
  switch (TWSR & 0xf8)
    {
    case 0x08:
      {
    if (read==0)
      TWDR = 0x44;//44
    else
      TWDR = 0x45; // read from slave
    TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    flagT=0;
    flagR=0;
```

```
    break;
      }
    case 0x18:
      {
    TWDR = dataT[flagT];
    TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    break;
      }
    case 0x28:
      {
    flagT++;
    TWDR = dataT[flagT];
    TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    break;
      }
    case 0x30:
      {
    TWCR = (1<<TWSTO) | (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    break;
      }
      // --------- Master receiver mode -------------------------
    case 0x40:
      {
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    break;
      }
    case 0x50:
      {
    dataR[flagR]=TWDR;
    flagR++;
    if (flagR<3) {
      TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE);
    } else {
      TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
    }
    break;
      }
    case 0x58:
      {
    dataR[flagR]=TWDR;
    TWCR = (1<<TWSTO) | (1<<TWINT)| (1<<TWEN) | (1<<TWIE);
    break;
      }
    default: break;
    }
}

volatile uint32_t hex=0;
uint8_t j=0,chan=0,sign=0,cmd=0;
uint16_t value=0;
char ch[MESSAGE_LENGTH];

SIGNAL (SIG_UART_RECV)//(USART_RXC)
{
  ch[j]=UDR;
  j++;

  if (j==8) {
    j=0;
    hex=convertStringToHex(ch);

    cmd = (hex & 0xf0000000)>>28;
    chan = (hex & 0x0ff00000)>>20;
    sign = (hex & 0x000f0000)>>16;
    value = hex & 0x0000ffff;

     switch (chan)
```

```
      {
      default: TWI_transmit_4Byte(hex); break;
      }
  }
}

void TWI_transmit_4Byte(uint32_t hex)
{
  int i=0;
  for (i=3;i>-1;i--) {
    dataT[i]=hex;
    hex = hex >> 8;
  }
  read = 0;
  TWCR = (1<<TWSTA) | (1<<TWINT) | (1<<TWEN) | (1<<TWIE);
}

int main(void){

  USART_Init();
  TWI_Init();

  while (1)
    {
    }
}
```

D. Source Code of the Etrax Program

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
//RS232
#include <fcntl.h>   /* File control definitions */
#include <errno.h>   /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */
pthread_t p1,p2,p4;

//common
#define MESSAGE_LENGTH 8
char msg[MESSAGE_LENGTH+1];
char msg_r[MESSAGE_LENGTH+1];

//RS232
char serial_in[MESSAGE_LENGTH+1];// one byte more!!
int sd, sd_2, n_2;

//UDP
#define LOCAL_SERVER_PORT 1500
int cliLen,n,fd_;
struct sockaddr_in cliAddr, servAddr, remoteServAddr, cliAddr_2;
int rc_2,servLen;
#define REMOTE_SERVER_PORT 1501

int NoRoom=0;

int light=0;
uint32_t temp=0,position=0;

void sendUDP(char *data);
void sendUDP_2(char *data);

waitAmom(int count) {
  int i=0,j=0;
  for (i=0;i<count;i++) {
    for (j=0;j<0xffff;j++)
      {}
  }
}

uint32_t convertStringToHex(char *string) {
  uint32_t i=0,fourbytes=0,value=0;
  for (i=0;i<MESSAGE_LENGTH;i++)
    {
      if ('0' <= *(string+i) && *(string+i) <='9') {
    value = *(string+i) - '0';
      }
      else if (('A' <= *(string+i) && *(string+i) <= 'F')) {
    value = *(string+i) - 'A' + 10;
      }
      else if (('a' <= *(string+i) && *(string+i) <= 'f')) {
          value = *(string+i) - 'a' + 10;
```

```
      }
      fourbytes = fourbytes << 4;
      fourbytes |= value;
    }
return fourbytes;
}

char* makeString(char *cmd, char *channel, uint32_t value)
{
  uint8_t i;
  uint32_t mask=0x000f0000;
  static char sendData[9];
  char hex[5];

  memset(&sendData, 0, sizeof(sendData));//important!!
  for (i=0;i<5;i++) {
    int v = (value & mask)>>(16-(i*4));
    if (0 <= v && v <= 9) {
      hex[i] = '0' + v;
    } else {
      hex[i] = 'A' + v - 10;
    }
    mask = mask >> 4;
  }

  sendData[0]=*(cmd+0);
  sendData[1]=*(channel+0);
  sendData[2]=*(channel+1);
  sendData[3]=*(hex+0);
  sendData[4]=*(hex+1);
  sendData[5]=*(hex+2);
  sendData[6]=*(hex+3);
  sendData[7]=*(hex+4);

  return sendData;
}

void RS232_send(char* out) {
  write(fd_,out,MESSAGE_LENGTH);
  printf("sendRS232: %s\n",out);
}

void *RS232_receive(char *fd) {
  char ch;
  int i, temp;
  while(1) {
    memset(&serial_in, 0, sizeof(serial_in));//important!!
    read(fd_,&serial_in,MESSAGE_LENGTH);
    i=strlen(serial_in);
    if (i<8)
      continue;
    printf("receivedSR232 %s %d\n",serial_in,i);
    //    sendUDP(serial_in);
    temp=convertStringToHex(serial_in);
    if ((temp & 0xfff00000)==0x00400000)
      position=temp & 0x000fffff;
   }
}

void *receiveUDP(int *ii) {
  cliLen = sizeof(cliAddr);

  while (1) {
    memset(&msg_r, 0, sizeof(msg_r));
    n = recvfrom(sd, msg_r, MESSAGE_LENGTH, 0,
         (struct sockaddr *) &cliAddr, &cliLen);
    if (msg_r[0]=='<') {
```

```
      if (strcmp(msg_r,"<<auto>>")==0) {
    sendUDP_2("ready...");
      }
      else if (strcmp(msg_r,"<<init>>")==0)
    { continue; }
      else if (strcmp(msg_r,"<room02>")==0)
    {
      NoRoom=2;
    }
      else if (strcmp(msg_r,"<<stop>>")==0)
    {
      light=0;
      NoRoom=0;
      RS232_send("10700000");
    }
      else if (strcmp(msg_r,"<<move>>")==0)
    {
      sendUDP_2("<<goon>>");
      RS232_send("107000ff");
    }
      else if (strcmp(msg_r,"<room01>")==0)//strncmp(msg_r,"107",3)==0)
    {
      NoRoom=1;
    }
      else if (strcmp(msg_r,"<light>>")==0)
    {
      light=light+1;

      if (NoRoom!=light)
        {
          RS232_send("107000ff");
          waitAmom(400);
          sendUDP_2("<<goon>>");
        }
    }
      else if (strcmp(msg_r,"<<back>>")==0)
    {
      RS232_send("101800af");
      RS232_send("10400000");
      position=0;
      while (position-5>(0xFFFFF-0x750-5))
        { waitAmom(5); RS232_send("00400000"); }
      waitAmom(5);
      sendUDP_2("<<goon>>");
      RS232_send("107000ff");
    }
      else if (strcmp(msg_r,"<<resL>>")==0)
    {
      light=0;
    }
    }
    else {
      printf("receivedUDP: %s %d\n",msg_r,strlen(msg_r));
      RS232_send(msg_r);//send it further via RS232 !!!
    }
  }
  return NULL;
}

void *GoToRoom(int *ii) {
  while (1) {
    switch (NoRoom)
      {
      case 0: break;
      case 1:
    {
      if (light==1)
```

```
        {
          RS232_send("107000ff"); // necessary??????
          RS232_send("10400000");
          while (position<0x3300)
        { waitAmom(10); RS232_send("00400000"); }
          waitAmom(10);
          RS232_send("10700000");
          waitAmom(10);

          // turn to door
          RS232_send("10400000");
          RS232_send("101000af");
          position=0;
          while (position<0x750)
        { waitAmom(5); RS232_send("00400000"); }
          waitAmom(5);
          RS232_send("10700000");
          NoRoom=0;
        }
      break;
    }
      case 2:
    {
      if (light==2)
        {
          RS232_send("107000ff");  // necessary???????
          RS232_send("10400000");
          position=0;
          while (position<0x4050)
        { waitAmom(10); RS232_send("00400000");}
          waitAmom(10);
          RS232_send("10700000");
          NoRoom=0;
        }
      break;
    }
      case 3: break; //nothing, cause no door
      case 4: break;
      }
  }
}

void sendUDP(char *data) {
  printf("sendUDP: %s\n", data);
  n = sendto(sd, data, MESSAGE_LENGTH, 0,
         (struct sockaddr *) &cliAddr, cliLen);
}

void sendUDP_2(char *data) {
  printf("sendUDP_2: %s\n", data);
  n_2 = sendto(sd_2, data, MESSAGE_LENGTH, 0,
        (struct sockaddr *) &remoteServAddr, sizeof(remoteServAddr));
}

int init_serial(char serial_port[]) {
  struct termios settings;

  int r2d2 = open(serial_port, O_RDWR);
  tcgetattr(r2d2, &settings);
  settings.c_iflag = 0;
  settings.c_oflag = 0;
  settings.c_lflag = 0;
  settings.c_cflag = CLOCAL | CS8 | CREAD;
  settings.c_cc[VMIN] = 1;
  settings.c_cc[VTIME] = 0;
  cfsetispeed(&settings, B38400);
  cfsetospeed(&settings, B38400);
```

```
  tcsetattr(r2d2, TCSADRAIN, &settings);
  return r2d2;
}

int main(int argc, char *argv[]) {

  int rc,rc_2;
  struct hostent *h;

  // socket creation
  sd=socket(AF_INET, SOCK_DGRAM, 0);
  if(sd<0) {
    printf("%s: cannot open socket \n",argv[0]);
    exit(1);
  }

  // bind local server port
  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servAddr.sin_port = htons(LOCAL_SERVER_PORT);
  rc = bind (sd, (struct sockaddr *) &servAddr,sizeof(servAddr));
  if(rc<0) {
    printf("%s: cannot bind port number %d \n",
       argv[0], LOCAL_SERVER_PORT);
    exit(1);
  }

  printf("%s: waiting for data on port UDP %u\n",
     argv[0],LOCAL_SERVER_PORT);

  // ------------------

  /* get server IP address (no check if input is IP address or DNS name */
  h = gethostbyname("192.168.0.1");//localhost");
  if(h==NULL) {
    printf("%s: unknown host '%s' \n", argv[0], argv[1]);
    exit(1);
  }

  printf("%s: sending data to '%s' (IP : %s) \n", argv[0], h->h_name,
     inet_ntoa(*(struct in_addr *)h->h_addr_list[0]));

  remoteServAddr.sin_family = h->h_addrtype;
  memcpy((char *) &remoteServAddr.sin_addr.s_addr,
     h->h_addr_list[0], h->h_length);
  remoteServAddr.sin_port = htons(REMOTE_SERVER_PORT);

  /* socket creation */
  sd_2 = socket(AF_INET,SOCK_DGRAM,0);
  if(sd_2<0) {
    printf("%s: cannot open socket \n",argv[0]);
    exit(1);
  }

  /*
  // bind any port
  cliAddr.sin_family = AF_INET;
  cliAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  cliAddr.sin_port = htons(0);

  rc_2 = bind(sd_2, (struct sockaddr *) &cliAddr, sizeof(cliAddr));
  if(rc_2<0) {
    printf("%s: cannot bind port\n", argv[0]);
    exit(1);
    }*/
```

```
  //-----------------------------


  fd_=init_serial("/dev/ttyS0");   // init serial port

  pthread_create(&p4, NULL, RS232_receive, &fd_);
  pthread_create (&p1, NULL, receiveUDP, &sd);
  pthread_create (&p2, NULL, GoToRoom, &sd);

  while (1)
    {

    }

  pthread_join (p4, NULL);
  pthread_join (p1, NULL);
  pthread_join (p2, NULL);

return 0;
}
```

E. Source Code of the Matlab Program

```
clear all;
clc

host='192.168.0.90';
port='1500';
udp=pnet('udpsocket',1501);

format short;

dat='        ';
while dat ~= 'ready...'
 try,
      % Wait/Read udp packet to reed buffer
      len=pnet(udp,'readpacket');
      if len>0,
        dat=pnet(udp,'read',1000,'string')
      end
  end
end


%------------------------------

patter=imread('/local/jens/scene6/init3','jpg');
pattern=patter(:,:,1);
pattern=double(pattern);

c2=0;
ind_sav=0;

T=0.7;
Tn=4;
Tv=0.1;
Kr=1;
u_old=0;
u=0;
e=0;
e_1=0;
e_2=0;
ind=0;
u_sav=0;

init_end=0;

while (init_end==0)
  clear I;
  I=imread('http://192.168.0.91/axis-cgi/jpg/image.cgi?resolution=160x120','jpg');
  I=I(:,:,1);
  I=double(I);%I=I-sum(sum(I))./19200;

  maxim=0;
  sav = 0;

  [Rows,Cols] = size(pattern);
  CorrCoeff = zeros(120-Rows+1,160-Cols+1);

  for i=1:3
    for j=10:160-Cols-1-10
      I_w = I(i:i+Rows-1,j:j+Cols-1);
```

```
    CorrCoeff(i,j) = corrco(I_w,pattern);
    if CorrCoeff(i,j)>0.995
      if (CorrCoeff(i,j)>maxim)
        maxim=CorrCoeff(i,j);
        ind=j;
      end
    end
  end
end

if (maxim>0.995)

ind_sav=[ind_sav ind];

if ((ind>=49) & (ind<=50))
  c2=c2+1;
  if c2>=4
    c2=0;
    disp('YEAHHHHHHHHH THE ROBOT IS CENTERED');
    dat='10700000';
    sendUDP(host,port,dat);
    init_end=1;
    for kkk=1:1:9000000
      hj=1+ind;
    end

  else
    dat='10000000';
    sendUDP(host,port,dat);
  end

else
  c2=0;
  e = 48 - ind;
  u = u_old + Kr*(1+Tv/T)*e - Kr*(1-T/Tn+2*Tv/T)*e_1 + Kr*Tv/T*e_2;

  if (e<0)
    if u>=-100
      u=-100;
    end
  else
    if u<100
      u=100;
    end
  end

  e_2=e_1;
  e_1=e;
  u_old = u;
  if u<0
    sign=0;
  else
    sign=8;
  end
    u=abs(u);
    u = round(u);

  if u>383;
    u=383;
  end
  if u<hex2dec('10')
    dat=['100',num2str(sign),'000',num2str(dec2hex(u))];
  elseif u<hex2dec('100')
    dat=['100',num2str(sign),'00',num2str(dec2hex(u))];
  else
    dat=['100',num2str(sign),'0',num2str(dec2hex(u))];
  end
```

```
    sendUDP(host,port,dat);
    dat='10100000';
    sendUDP(host,port,dat);
  end

  else
    dat='10000070';
    sendUDP(host,port,dat);
    dat='10100070';
    sendUDP(host,port,dat);
  end
end
%----------------------------



clear all;

host='192.168.0.90';
port='1500';
udp=pnet('udpsocket',1501);

format short;
command='        ';
dat='        ';

patter=imread('/local/jens/scene6/light2','jpg');%'/local/jens/160x120_spec_1','jpg');
pattern=patter(:,:,1);
pattern=double(pattern);

c2=0;
ind_sav=0;

T=0.8;
Tn=1;%1.0
Tv=0.1;%0.1
Kr=1;%1.0
u_old=0;
u=0;
e=0;
e_1=0;
e_2=0;
ind=0;
u_sav=0;


for c=0:2000
  clear I;
  I=imread('http://192.168.0.91/axis-cgi/jpg/image.cgi?resolution=160x120','jpg');

  I=I(:,:,1);
  I=double(I);%I=I-sum(sum(I))/19200;

  maxim=0;

  [Rows,Cols] = size(pattern);
  CorrCoeff = zeros(120-Rows+1,160-Cols+1);

for i=1:3
  for j=10:160-Cols-1-10
    I_w = I(i:i+Rows-1,j:j+Cols-1);
    CorrCoeff(i,j) = corrco(I_w,pattern);
    if CorrCoeff(i,j)>0.94
      if (CorrCoeff(i,j)>maxim)
        maxim=CorrCoeff(i,j);
```

```
        ind=j;
      end
    end
  end
end

if (maxim>0.94)

  ind_sav=[ind_sav ind];

  if ((ind>=50) & (ind<=54))
    c2=c2+1;
    if c2>=4
      c2=0;
      disp('YEAHHHHHHHHH THE ROBOT IS CENTERED');
      dat='<light>>';
      sendUDP(host,port,dat);

      command=='<<wait>>';
      try
        len=pnet(udp,'readpacket');
        if len>0,
          command=pnet(udp,'read',8,'string');
        end
      end

      command

      if command=='<<goon>>'
        continue;
      end
      if command=='<<exit>>'
        return;
      end

      else
        dat='10000000';
        sendUDP(host,port,dat);
      end

    else
      c2=0;
      e = 52 - ind;
      u = u_old + Kr*(1+Tv/T)*e - Kr*(1-T/Tn+2*Tv/T)*e_1 + Kr*Tv/T*e_2;

      if (e<0)
        if u>=-90
          u=-90;
        end
      else
        if u<90
          u=90;
        end
      end

      e_2=e_1;
      e_1=e;
      u_old = u;
      if u<0
        sign=0;
      else
        sign=8;
      end

      u=abs(u);
      u = round(u);
      if u>383;
```

```
        u=383;
      end
      if u<hex2dec('10')
        dat=['100',num2str(sign),'000',num2str(dec2hex(u))];
      elseif u<hex2dec('100')
        dat=['100',num2str(sign),'00',num2str(dec2hex(u))];
      else
        dat=['100',num2str(sign),'0',num2str(dec2hex(u))];
      end
      sendUDP(host,port,dat);

      dat='10100000';
      sendUDP(host,port,dat);
    end

  else
    if dat~='107000bf'
      dat='107000bf';
      sendUDP(host,port,dat);
    end
  end
end


function sendUDP(host, port, data)

if udp~=-1,
  try, % Failsafe
      pnet(udp,'write',data);              % Write to write buffer
      pnet(udp,'writepacket',host,port);   % Send buffer as UDP packet
  end
end

end
```

F. Source Code of the Java-applet

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
import java.io.IOException;
import java.net.URL;
//Socket
import java.net.*;
import java.io.*;

import java.lang.String;

public class r2d2 extends JApplet implements ActionListener,MouseListener,
      MouseMotionListener {
  DatagramSocket socket=null;
  String host ="192.168.0.90";   // Adress Etrax
  final static boolean shouldFill = true;

  Navigator MouseNavigator;

// invoked after the visible components are displayed
public void init() {
  try {
  javax.swing.SwingUtilities.invokeAndWait(new Runnable() {
  public void run() {
    createGUI(getContentPane());
  }
    });
  } catch (Exception e) {
  System.err.println("createGUI didn't successfully complete");
  e.printStackTrace();
}
initUdp();
}

private void createGUI(Container pane) {
  pane.setLayout(new GridBagLayout());
  GridBagConstraints c = new GridBagConstraints();
c.insets = new Insets(1,1,1, 1);
if (shouldFill) {
    // natural height, maximum width
    c.fill = GridBagConstraints.HORIZONTAL;
  }

  MouseNavigator = new Navigator();
  MouseNavigator.setPreferredSize(new Dimension(200, 200));
  MouseNavigator.setBorder(BorderFactory.createLineBorder(Color.black));
  c.gridx=1;
  c.gridy=0;
  c.gridheight = 3;
  pane.add(MouseNavigator,c);c.gridheight = 1;

  MouseNavigator.addMouseListener(this);
  MouseNavigator.addMouseMotionListener(this);
}

public void actionPerformed(ActionEvent event) {
}
```

```
public void initUdp() {
  try {
  socket = new DatagramSocket();
    // send request
  byte[] buf = new byte[8];
  InetAddress address = InetAddress.getByName(host);
  String dString="<<init>>";
  buf=dString.getBytes();
  DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 1500);
  socket.send(packet);

  // thread waits for incomming udp packets
  input_thread it = new input_thread(socket);
  it.start();
  }
  catch (IOException e) {
  e.printStackTrace();
}
}

public void SendMessageViaUdp(String Cmd, String Channel, int Value, int sign) {
  String CmdChannelValue=Cmd;
  String strValue;

  CmdChannelValue = CmdChannelValue.concat(Channel);
  strValue = Integer.toString(Value,16);

  if (sign==1) {
  switch (strValue.length())
      {
      case 1: CmdChannelValue=CmdChannelValue.concat("8000"); break;
  case 2: CmdChannelValue=CmdChannelValue.concat("800"); break;
  case 3: CmdChannelValue=CmdChannelValue.concat("80"); break;
  case 4: CmdChannelValue=CmdChannelValue.concat("8"); break;
  }
  }
else {
    switch (strValue.length())
  {
  case 1: CmdChannelValue=CmdChannelValue.concat("0000"); break;
  case 2: CmdChannelValue=CmdChannelValue.concat("000"); break;
  case 3: CmdChannelValue=CmdChannelValue.concat("00"); break;
  case 4: CmdChannelValue=CmdChannelValue.concat("0"); break;
  }
}
  CmdChannelValue = CmdChannelValue.concat(strValue);

int MESSAGE_LENGTH=8;
byte[] buf = new byte[MESSAGE_LENGTH];
buf = CmdChannelValue.getBytes();
try {
  InetAddress address = InetAddress.getByName(host);
  DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 1500);
  socket.send(packet);
}
catch (IOException e) {
  e.printStackTrace();
}
}

public void mousePressed(MouseEvent e) {
}

// A method from the MouseMotionListener interface. Invoked when the
// user drags the mouse with a button pressed.
public void mouseDragged(MouseEvent e) {
  double height = MouseNavigator.getSize().height;
```

```
  double width = MouseNavigator.getSize().width;
  double x = (e.getX() - width / 2);// / width * 200;
  double y = (e.getY() - height / 2);// / height * 200;
  double s1 = Math.sqrt(0.5) * x + Math.sqrt(0.5) * y;
  double s2 = Math.sqrt(0.5) * x - Math.sqrt(0.5) * y;

  int IntCast1;
  int IntCast2;
  int sign1=0,sign2=0;
  IntCast1=(int)(s1*7 );
  IntCast2=(int)(s2*7 );
  if (IntCast1>511 )
    IntCast1 =511 ;
  if (IntCast2>511 )
    IntCast2 =511;
  if (IntCast1<-511 )
    IntCast1 =-511 ;
  if (IntCast2<-511 )
    IntCast2 =-511 ;

  if (IntCast1<0) {
    sign1=1;
    IntCast1 =Math.abs(IntCast1);
}
  if (IntCast2<0) {
    sign2=1;
  IntCast2 =Math.abs(IntCast2);
}

SendMessageViaUdp("1","00",IntCast1, sign1);
for (int u=0;u<699999;u++) { int k=0;}
SendMessageViaUdp("1","01",IntCast2, sign2);
for (int u=0;u<6999999;u++) { }
}

public void mouseReleased(MouseEvent e) {
// stop
SendMessageViaUdp("1","00",0,0);
SendMessageViaUdp("1","01",0,0);
}

// The other, unused methods of the MouseListener interface.
public void mouseClicked(MouseEvent e) {;}
public void mouseEntered(MouseEvent e) {;}
public void mouseExited(MouseEvent e) {;}

// The other method of the MouseMotionListener interface.
public void mouseMoved(MouseEvent e) {;}

}
```