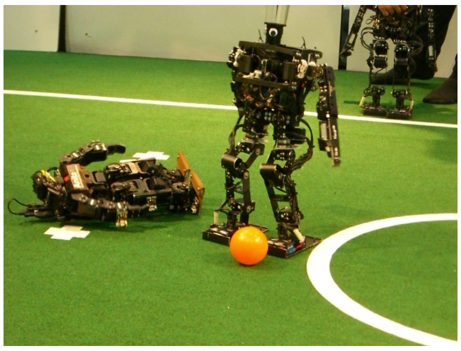
Universität Dortmund

Project group 499 final report

Biped Soccer Robots
Development of a Universal Robotic Software and Hardware Architecture



Robotics Research Institute

PG Participants:
Lamia Chouaieb, Nikolas Ehrenberg, Niklas Goddemeier, Daniel Hauschildt, Jens Hegenberg, Daniel Klagges, Simon Niedzwiedz, Boris Schauerte, Robert Schmidt, Patrick Szcypior, Christopher Trampisch, Jens Walkenhorst, Adalbert Wilczek

Advisors:
Prof. Dr.-Ing. Uwe Schwiegelsohn
Matthias Hebbel
Walter Nistico

Date:
November 5, 2007

Contents

Introduction

This paper describes the activities of the project group 499 at the Dortmund University. The main aim of this project group was to create a walking, soccer playing humanoid robot. With the use of this robot the participation in the Robocup German Open 2007 and the Robocup Atlanta 2007 was the final goal. The German Open is the german (european) championship and the Robocup Atlanta is the world championship in robot soccer organized by the Robocup Federation[1]. The aim of this federation is to support artificial intelligence and robotics research. A "future goal" is the development of autonomous robots being able to compete against the human World Soccer Champion in the year 2050.
The Robocup Soccer is splitted up in several leagues: a simulation league, different leagues for wheeled robots (small size, middle size) and a four legged league using the AIBO entertainment robot. There are also two leagues for humanoid robots: the kid size league and the teen size league.
The BreDo Brothers, a robots soccer team collaboration of Bremen University and Dortmund University, were playing in the kid size league in 2006. The BreDo Brothers Team used a Kondo KHR-1 humanoid robot. Due to restrictions of this robot, it was decided to create a similar humanoid robot with more competitive specifications. The new robot is also a kid size robot. The developement of this robot is being done by the University of Dortmund. The former team split up.
This document gives an overview over the different topics the project group worked on, including electronics (Chapter 1), hardware (Chapter 2) and software parts (Chapter 3 and 4). Some of the parts described in this document are further developments or derivatives of the preceding project groups at the University of Dortmund (No. 485 and No. 468). *Robot Control XP* control and other debugging software was reused, for example. Others are redesigns of existing parts, e.g. the robot framework or the robot body design. But many parts are completely new as they were needed for the first time, as the boards like the embedded board, the carrier boards or the security board. Also, sensors like the gyroscope and the acceleration sensor were completely new.

It was tried to pick the best possible parts in every area for the new robot. Thus a strong dual core cpu powered board is used in connection with an omnivision camera for fast and reliable computer vision. The omnivision camera system uses computer vision based on the diploma thesis of Stefan Czarnetzki who also implemented the needed software [Cza07]).
Microsoft Windows XP Embedded was chosen as operating system (see chapter 3).

1 www.robocup.org

As a hip joint was missing in the preceding robot (Kondo KHR-1)[KKC04] the robot body has been redesigned, now consisting of modular brackets by *lynxmotion*[2] and servos by *Hitec*[3]. The body design was redesigned several times during the project group's year. The design steps are discussed in detail in chapter 2.
This year, *Microsoft Robotics Studio* was used for the first time instead of the self developed simulator used before (see chapter 4.1). Due to shortcomings in the old robot framework, the framework has been redesigned (see chapter 4.2). With a new robot body design, new motion patterns needed to be developed. See chapters 4.4, 4.6 and 4.7 for a description on that.
Due to the many changes made in the framework, sensors and other parts, also the robot control tool had to be modified (see chapter 4.9).

2 http://www.lynxmotion.com

3 http://www.hitecrc.de

1 Electronics

When developing a new two legged robot platform, it has to be decided what kind of processing power is needed for the robot. Since most current humanoid robots are limited in their processing power the choice was made to have something more convenient. The problem that arises is that only limited space for the processing unit is available. So only certain form factors can be used. The decision was made that a XTX-Module, a successor of ETX-Modules, would be the best choice due to its good performance to size ratio. It also contains several components that can be found in a standard personal computer.

1.1 Embedded Board

The module chosen was a XTX board manufactured by Congatec[4] called *Conga-X945* (see table 1.1 for specifications).

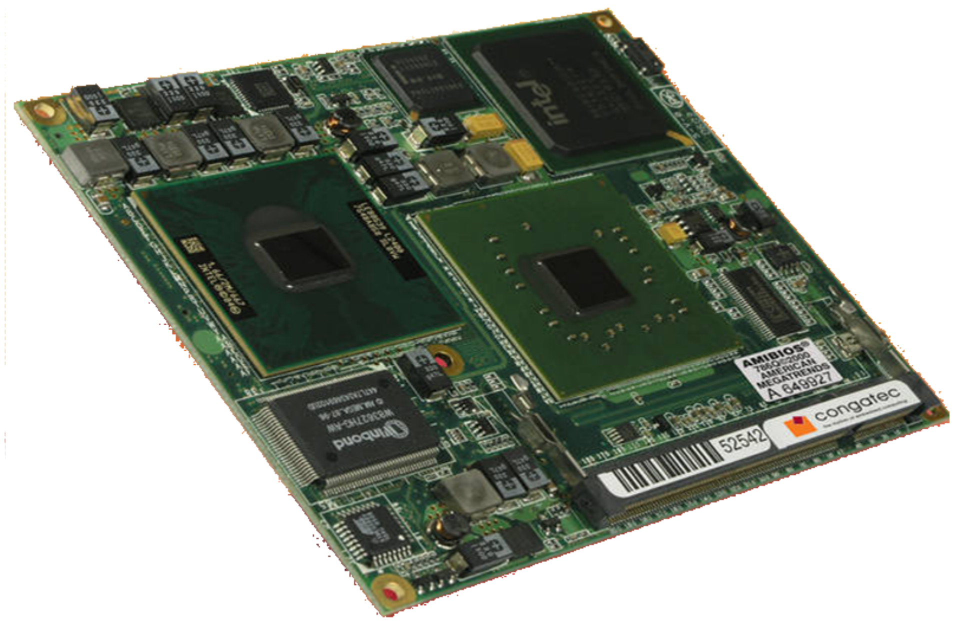


Figure 1.1: Conga-X945

This module was chosen because all needed features were available (e.g. USB 2.0 for

4 http://www.congatec.com

Processor	Intel Core Duo L2400 1.66Ghz, 2-MByte L2 cache LV[a]
Memory	512 MB S0-DIMM DDR2 667
Chipset	GMHC[b] Intel 82945GM and ICM[c] Intel 82801GBM
Audio	Realtek ALC 655 AC'97 Rev. 2.2
Ethernet	ICH7M with PHY[d] Intel 82562
Graphics	Intel GMA[e] 950 with max. 224 Dynamic Memory
Super I/O	Winbond 83627HG
Peripheral Interfaces	2x Serial ATA, 4x 1x PCI Express, 6x USB 2.0, 1x IDE, PS/2 Keyboard + Mouse, I2C Fast Mode 400 KHz multi-master, 2x COM Ports TTL Level, AC'97/HDA[f] Digital Audio Interface
BIOS	Based on AMIBIOS8-1MByte Flash Bios with congatec Embedded BIOS features

a low voltage

b Graphics and Memory Controller Hub

c I/O Controller Hub

d Physical Interface

e Graphics Media Accelerator

f High Definition Audio codecs

Table 1.1: Conga-X945 Specifications

the cameras). Also, the power consumption is very low. Since this module is only a "baseboard" with basic functionality, some kind of carrier board (see section 1.2.1) and a suitable power-supply (see section 1.4), to power the board, had to be built.

1.2 Carrier Board

The *carrier board* is the parent-board for XTX modules. It was specially designed for the specific XTX-module. After the XTX-module was chosen, it had to be figured out which features of the board were needed and which additional features had to be integrated on the board. After three months of development the first revision(see section 1.2.1) was finished. After another three month the second revision (see section 1.2.2) was finished and being used during the *RoboCup 2007* in Atlanta.

1.2.1 Revision 1

The first revision was a simple design, only consisting of really necessary and easy to handle components. So the first revision (see section 1.2 mainly consisted of four USB 2.0 ports, four I2C device ports, 2 serial communication ports, one +5V DC power connector, a proprietary debug port, a SATA connector and an 100 Mbit ethernet connector. The +5V power supply was chosen to be an extra board for the first revision, Wireless LAN connection was realized via an additional USB 2.0 device. The connection to the servos was established via an additional board that was connected to a RS232 serial interface.

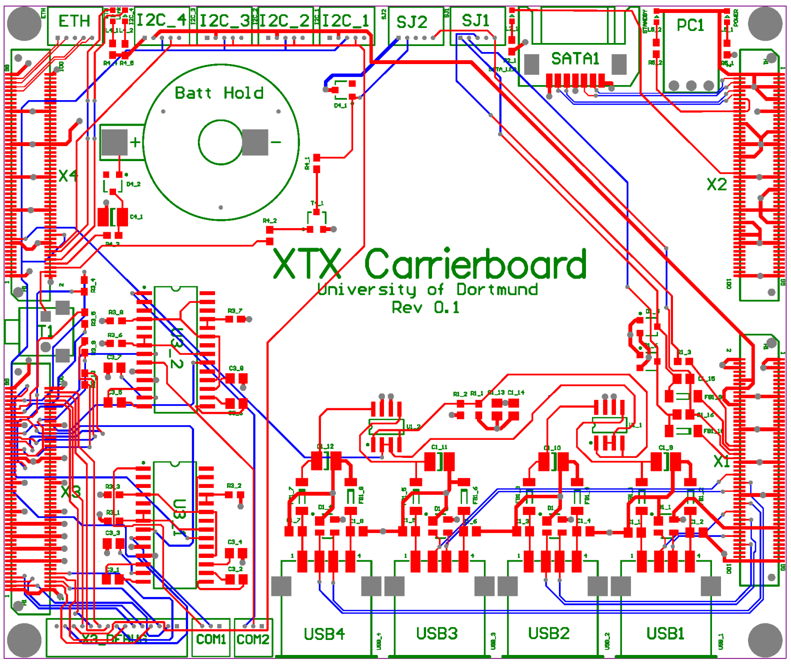


Figure 1.2: Carrier Board Revision 1.0

- **USB 2.0**
  When designing a robot it is necessary to keep in mind that several peripherals (e.g. the cameras) which would require high bandwidth have to be connected. For example, the desired resolution for the camera images was 1024x768 pixels. These images will be transfered as 8 bpp [5] Bayer Pattern with a framerate of about 25 fps[6]. So the camera needs at least a bandwidth of approximately 150 Mbit/s. To solve the bandwidth requirement problems it was decided to use *USB 2.0* with an approximate bandwidth of 480 Mbit/s. The *XTX embedded board* already had an *USB 2.0*-Controller on-board. In addition, there are plenty of peripherals available on the consumer market that support *USB 2.0*. Since *USB 2.0* is an high speed differential bus it is important to keep the layout consideration in mind when designing and routing the PCB[7]. All design criterias given by the USB 2.0 platform design guide were followed. A protection circuit(see figure 1.3) for the *USB* was also added which consists of a *Surge Rated Diode*[SEM], which protects

5 Bits per pixel

6 frames per second

7 printed circuit board

against overvoltage, and an *Dual-Channel Power Distribution Switch*[MIC], which protects against overcurrent and thermal destruction.

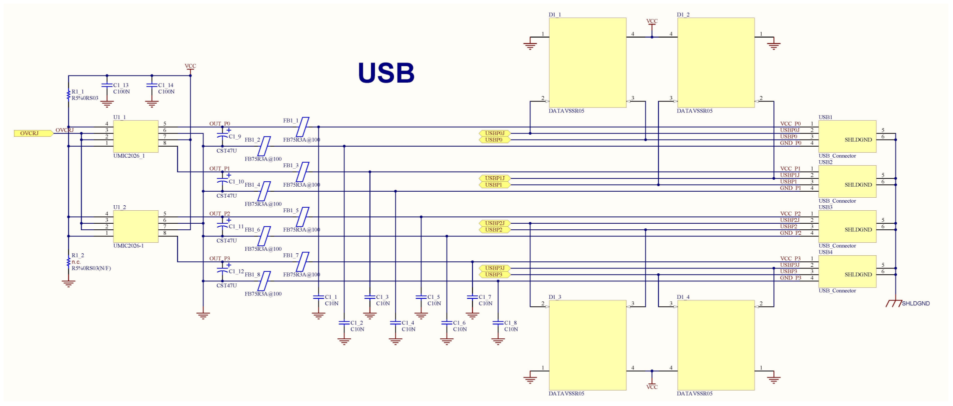


Figure 1.3: USB circuit with protecion

- **I2C bus**
  Since the *XTX embedded board* provided this bus it was decided to use it to interface the sensors. The 400Khz-*I2C bus* is able to interface at least three sensors (2 gyrometers and 1 acceleration sensors). With an approximate payload data rate of about 25 KByte/s and 2 Bytes per sensor a sample rate of approximately 4 KSamples/sensor seemed to be possible. Unfortunately, this was far from what later experiments should reveale. After some tests it was discovered that only about 20 samples per sensor could be reached. This is due to a bug in the *XTX embedded board* firmware which was already known by the vendor but not yet being fixed.

- **Wired/wireless communication**
  For wired communication the standard on-board 100Mbit ethernet controller is used. Since an ethernet plug was too big to be included on the carrier board, a custom connector [Mol] is used. Wireless communication is not directly integrated since it was much easier to just use an USB Wireless LAN stick instead of integrating another circuit on the board itself.

- **Serial communication**
  The embedded board includes two RS232 ports(see figure 1.4), for that the UART ports supplied by the XTX board are used together with RS232 transceivers which handle the conversion between serial port TTL level (+5V) and RS232 (+12V). Transceivers from *Analog Devices - ADM213EARZ* [Dev06] are used to fulfill this task. One RS232 port is dedicated for the communication to a previously developed servo motor controller. The other is designed to aid debugging the hardware. As for the wired communication normal RS232 D-Sub connectors, used in standard pc hardware, are too large so 3-pin mini-Molex [Mol] connectors are used for the serial ports to keep the dimensions of the board as small as possible.

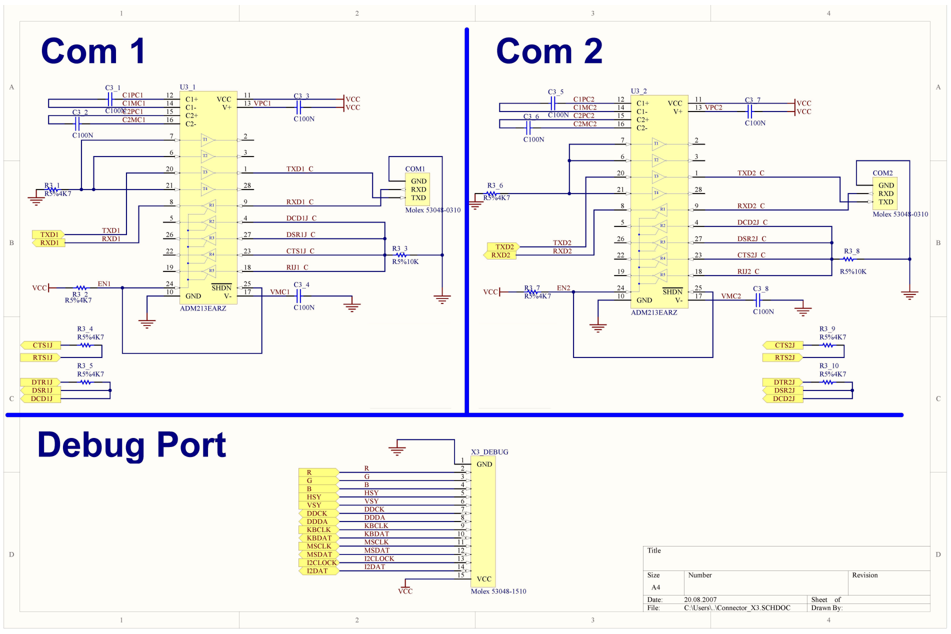


Figure 1.4: Serial port and Debug circuit

- **Debugging**
  Debugging was an really important point during the development. In order to interact with the operating system, VGA, keyboard and mouse support is included. Since this is not needed in normal operation, all required hardware is outsourced to an additional *debug board*(see section 1.3). A 16 pin mini-Moles connector[Mol] on the carrier board carries out all needed signals, for instance the ps/2 signals, I2C signals and VGA signals. Besides the debugging-connector, a *serial-ATA* port is included which allows the use of standard hard disks during development of the *Windows XP Embedded* operating system.

Evaluation

After using the first revision it was realized that some mistakes were made. The *mini-Molex*[Mol] connectors were not as good as they were expected to be. The connectors were too small and once they were connected it was hard to disconnect them. Also, the more the connectors were used, the more they were damaged. As one result of that, the wires broke out of the connectors. Beside the *mini-Molex* connectors the usb connector positions were not well reachable. Flexible *USB* extension cables were needed to use the ports. The *ethernet* connector was not working due to pcb routing mistakes. The *USB* connectors lagged some capacitors that were added afterwards by hand. Another problem that was not being thought of was how to deal with problems of the power supply. So no over-voltage or over-current protection was added to the board main-power supply, which turned out to be one of the biggest mistake, since two *embedded boards* were damaged due to power failures or misconnections. All this lead to the development of the second revision (see section 1.2.2) and the security board (see section 1.5).

1.2.2 Revision 2

After some tests with the first revision of the carrier board (see 1.2.1), it was recognized that some modifications had do be done to enhance the system. The first revision worked fine but was not very applicable for the work with the robot. The robot has to carry various additional devices like the power supply and the servo controller. Also, some parts like the *USB* stick and WLAN stick have to be connected in an uncomfortable way. To improve the old carrier board a second revision was necessary. This version of the carrier board should also be the final revision because of time and resource considerations. It was decided to order three boards, two for the robots and one as a spare part.

Additional features

In contrast to the first revision of the board there are some additional functionalities.

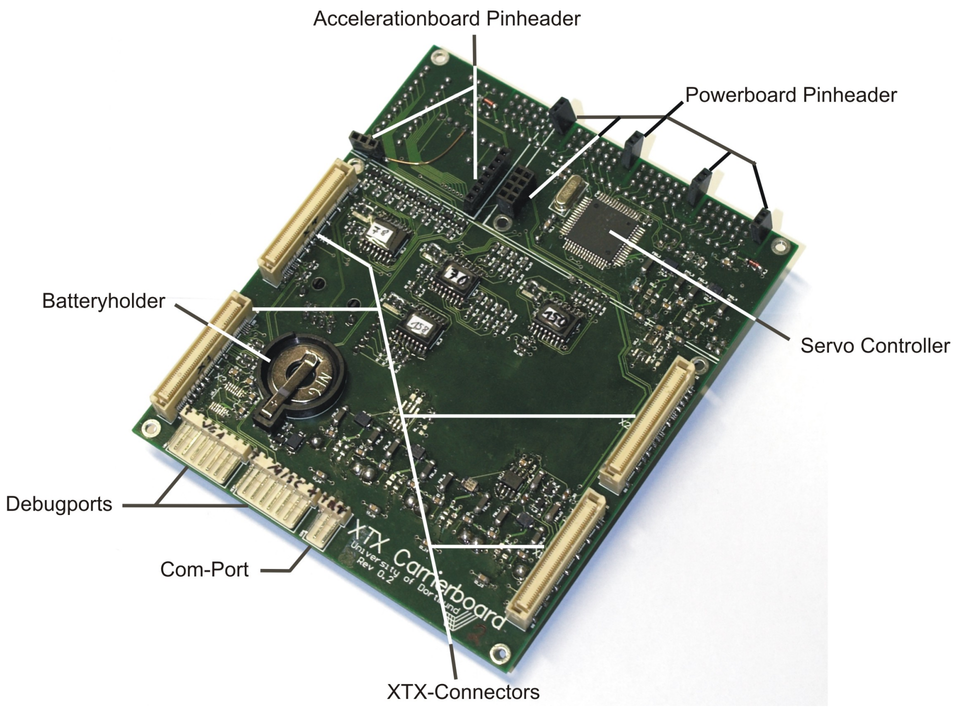


Figure 1.5: Front-Side of the carrier board

- **I2C-IO devices**
  Two new I2C devices were implemented with the general I/O expander *PCF8574* [Sem97] from *Philips*. One is connected as an input device which can be used for external buttons. The second is connected as an output device which can be used for LED debugging. Both devices have 8 inputs respectively outputs and can be

seen in figure 1.7.

- **I2C-AD Converter**
  Two built in AD converters are connected to the I2C bus usable for the gyroscopes (see 1.7.1). This devices are implemented with *PCF8591* [Sem98] ICs produced by *Philips*.

- **Servo-Controller**
  This controller is fully integrated to the carrier board (see figure 1.5). It consists of an *Atmega128* micro controller from *Atmel* [Cor06], a direct connection to the RS232 interface of the XTX board (see 1.1) and additional components needed for a proper operation. The device is generating the PWM signal for the servos (see 2.1). For that functionality it is connected to the 24 pin headers where the servos are connected to the carrier board (see figure 1.7).

- **Ethernet-socket**
  This standard ethernet socket [Inc] (see figure 1.7) is added to the carrier board. This enhances the error diagnostics if WLAN problems occur. It turned out that this additional socket is very useful especially on contests where WLAN is not working properly or simply is not available.

- **Two fan connectors**
  One is used for the CPU fan, the other is yet unused and reserved for further

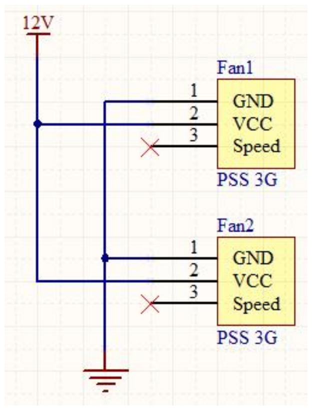


Figure 1.6: Fan-Connectors

  applications. Both connectors (see figure 1.7) are pin compatible with normal fan connectors (see figure 1.6) used in consumer electronics.

- **Additional I2C sockets**
  Different additional devices can be connected via this connectors. Some additional

sockets (see figure 1.7) are added to simplify the connection of devices for example the foot sensors.

- **Additional USB sockets**
  In the first revision it appeared that the USB sockets are to close together to really connect four devices at a time. Especially the WLAN stick is far to big and blocks at minimum two slots. For that reason two more USB slots are added to the board. These additional slots are no standard USB sockets but normal pin headers (see figure 1.7 - USB ports 5 and 6). The two cameras are connected to these pin headers. Later it turned out that the connection was not as fail safe as supposed so the USB cables where soldered directly to the pin headers.

- **Pin headers for connection of the acceleration board**
  An I2C acceleration sensor is added to the system to detect the orientation in space. This is very important for some special actions (see 4.4) like the stand up movement (see 4.4.2). The acceleration sensor *SMB 380* from Bosch is used which is able to measure the acceleration in three dimensions. This sensor is connected to the board via these additional pin headers (see figure 1.5) and is driven with I2C signals.

- **Several LEDs**
  Some status LEDs signalizing the status of the board and its periphery including whose ethernet connection. They have the same functionality as status LEDs on normal PC mainboards.

- **Redesign of debug connector**
  The debug connector of the first revision of the carrier board appeared to be to tiny. It got damaged to often. For that reason it was switched to a new more robust connection type (see figure 1.5). Apart from that no changes are made due to hardware debugging.

Changes to first revision

It became apparent to be useful to redesign some functionalities of the carrier board, while working with the first revision. While proceeding some tests of the first revision, some minor and major changes had to be made. The reasons and the resulting changes to the carrier board are listed below:

- **Simpler RS232 interface**
  The signal generation for the RS232 interface is simplified with the use of a *MAX3221*, which is a RS232 transceiver [Pro99]. There is a direct connection of the RS232 interface to the micro controller for the servos which makes the communication more fail safe. The second RS232 interface is routed to the border of the board (see figure 1.5) for directly debugging.

- **Layout changes**
  Some layout changes are made to simplify the connection of external devices. The

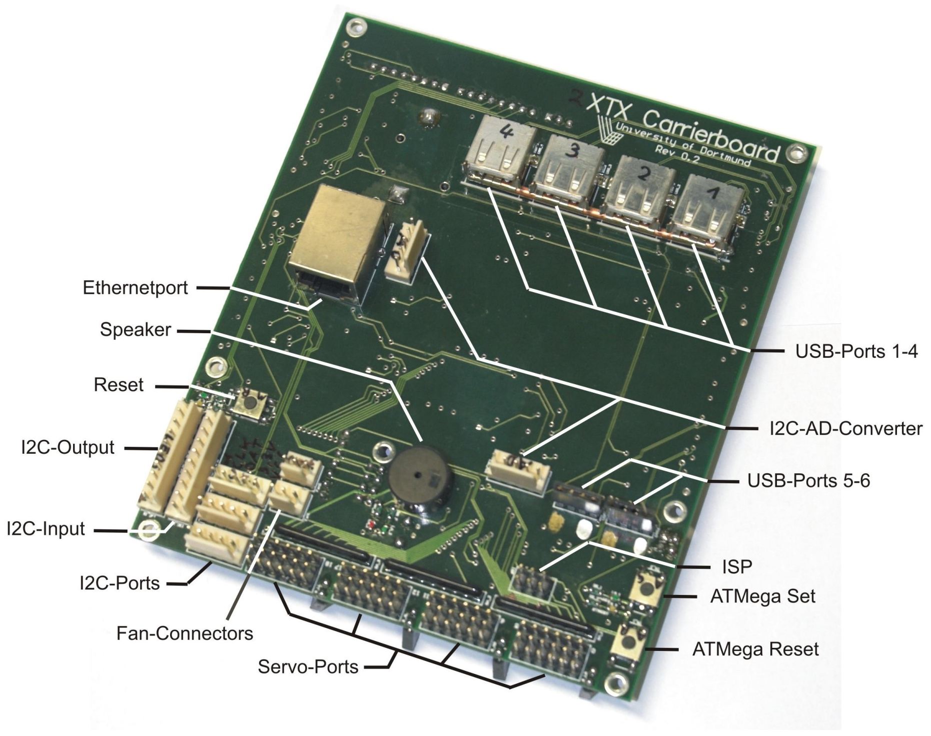


Figure 1.7: Backside of the carrier board

main change is that the USB ports are now placed inverted on the board (see figure 1.7) which allowed to connect the USB and WLAN stick without a cable behind the back of the robot, so that they are not in the view of the omni directional camera (see [Cza07]). The second main change is the usage of the sandwich technique, which will be described in 1.4.1.

- **No SATA interface**
  The SATA interface of the first revision is not implemented any more because it was never used. It is useless because the robot will never carry a hard drive.

1.3 Debug Board

While designing the carrier board it was noticed that it is necessary to have some kind of debug output. For the sake of compatibility, standard in- and output interfaces are used. These are a PS2 interface for keyboard and mouse inputs as well as a VGA interface for video signals. The realization of these interfaces was a relative easy task because all of the interfaces are implemented on the XTX board (see 1.1). Due to space considerations these interfaces are not directly implemented on the carrier board, but on a specific debug card, which holds all desired functionalities. Additionally to the features mentioned above, another I2C interface and a power LED are placed on that board.

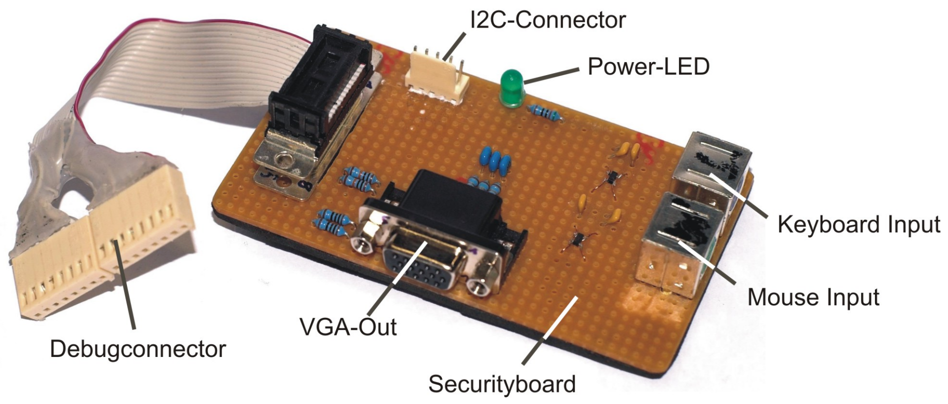


Figure 1.8: Debugboard

The circuitry of the debug board is very simple (see figure 1.8). There are only some resistors and capacities filtering noise away from the VGA connector and a over current protection for the keyboard and mouse interface (similar to the implementation of the over current protection for the USB ports in 1.2.1). The complete device is so robust that it can be disconnected from the robot, while it is in operation.

1.4 Power Board

A very important hardware capability is to supply servos and mainboard with power. To develop a power supply, working under the given weight and space constraints, turned out to be a complicated task. During some early tests with the carrier board's first revision (see 1.2.1) it appeared that the XTX board (see 1.1) consumes up to 5 amperes while booting. Calculations of the electric power consumption of the servos show a current need up to 20 amperes (worst case assumption). While in operation, especially during extreme motions of the robot (for example the stand up motion mentioned in 4.4.2), a load of approximately 10 amperes is generated. Linear voltage controllers are not useful due to their dissipation loss and as a result of that, their thermal leakage.

1.4.1 Sandwich Technique

The connection of the power supply and the XTX board (see 1.1) became a mentionable design criteria. It is possible to implement the power supply circuits directly on the carrier board (see 1.2.2), but concerning the fact that the power supply often gets damaged, a different approach was chosen. The power supply is mounted to the carrier board using pin headers. In this way a modular design called sandwich technique is achieved, which allows the power supply to be replaced easily. Another benefit of this sandwich technique is a decrease in required space. Later this sandwich technique is extended by the security board (see 1.5). In the end a very compact power supply with different fail safe functions is produced.

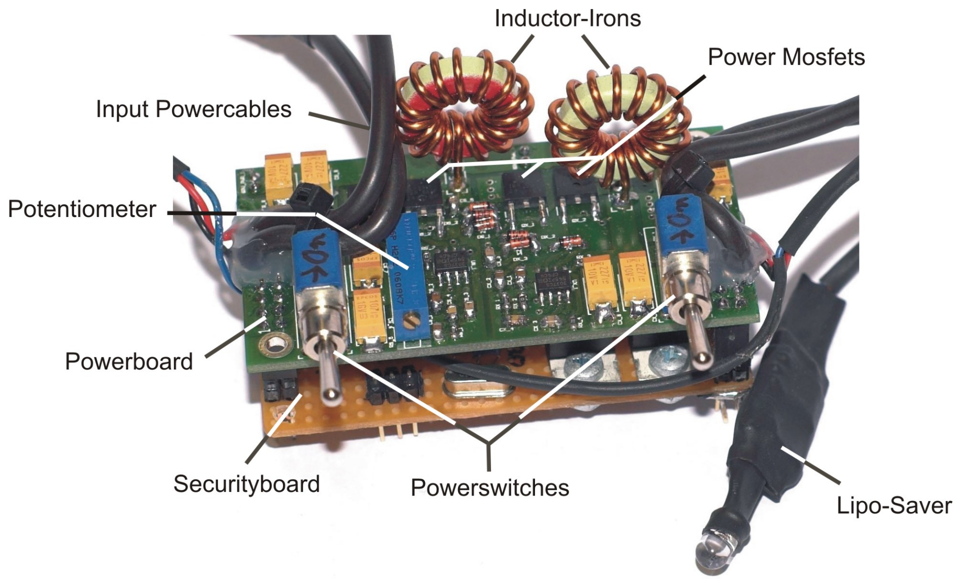


Figure 1.9: Power board and security board in sandwich technique

1.4.2 Power Board Operation

The main component of the power board is a synchronous PWM controller integrated into one IC. PWM means that the output power transistors are driven with pulse width modulation. A closed loop of the output voltage to the controller guarantees a stable voltage for different loads. The loop back is working with a frequency of 200kHz. It is used a *IRU3037* [Rec05] IC in combination with *IRLR3717* [Rec04] power mosfets from International Rectifier[8]. The *IRU3037* provides some useful additional functionalities like an external programmable soft start function as well as an output under voltage detection that latches off the device when a short circuit is detected on the output.
The *IRU3037* is generating the PWM signal with the HDrv and LDrv pins for the two power mosfets QX_1 and QX_2 deciding whether power is put on the output or not. Behind the transistors an inductor iron, with several turns and two parallel capacities CX_8 and CX_9, is used to stabilize the output voltage. The value of the output voltage is programmed by reference voltage and external voltage divider consisting of RX_3 and RX_4. The formula for that is:

$$V_{OUT} = V_{REF} \cdot \left(1 + \frac{RX\_3}{RX\_4}\right) \tag{1.1}$$

With the values $1k\Omega$ for RX_3 and 330Ω for RX_4 and an internal reference voltage $V_{Ref} = 1,25V$ (see data sheet [Rec05]) this is:

8 http://www.irf.com

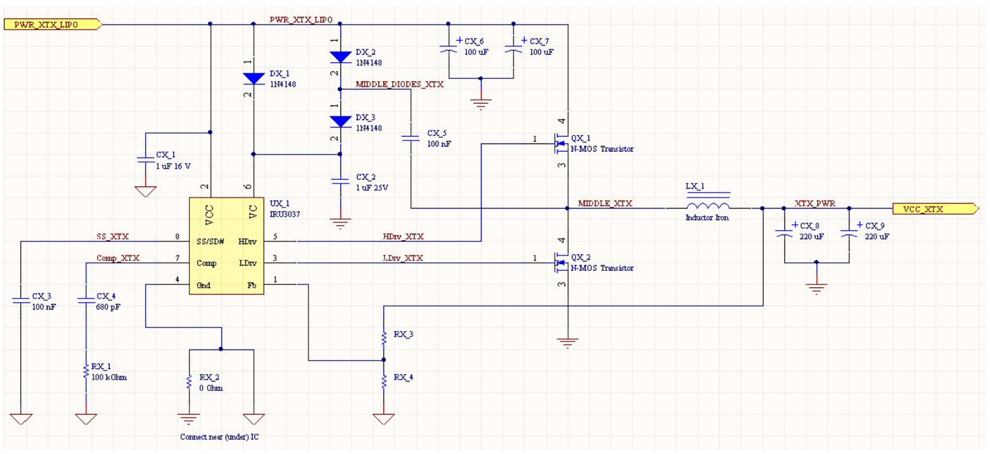


Figure 1.10: Schematic of power board

$$
\begin{aligned}
V_{OUT} &= V_{REF} \cdot \left(1 + \frac{1000\Omega}{330\Omega}\right) \\
V_{OUT} &= 5,04V
\end{aligned}
\qquad (1.2)
$$

In this type of application it is very important to use highly accurate resistor values. So only resistors with 1% tolerance are used. For the second power supply circuit, powering the servos, a potentiometer (see data sheet [Dep]) is used instead of the two fixed resistors which allows to create different output voltages.

1.5 Security Board

In process of the project, it turned out that it is important to monitor the power supply system (see 1.4) for security reasons. This relates to the input voltage, which is provided by the lithium polymer accumulators (see 2.2), as well as to the output voltage used for the XTX board (see 1.1) and the servos (see 2.1). Because of the fact that lithium polymer accumulators are damaged physically, when their voltage falls below a certain level (deep discharge), LiPo[9] savers are used, which measure the voltage level continuously. The LiPo savers provide LEDs, which signalize under voltage occurrences optically. Applied to the robot this security mechanism was not very practicable because the flashing of the LED can be easily ignored. As an advancement an acoustical warning in combination with the LEDs is implemented. This feature is placed on the security board.
The second important functionality of the security board is the protection of the servos and even more important the XTX board (see 1.1). They have to be protected against

9 Lithium polymer accumulator

over voltage, which would destroy them. To achieve this two crowbar circuits[10] are designed and implemented.

1.5.1 Acoustical LiPo Saver

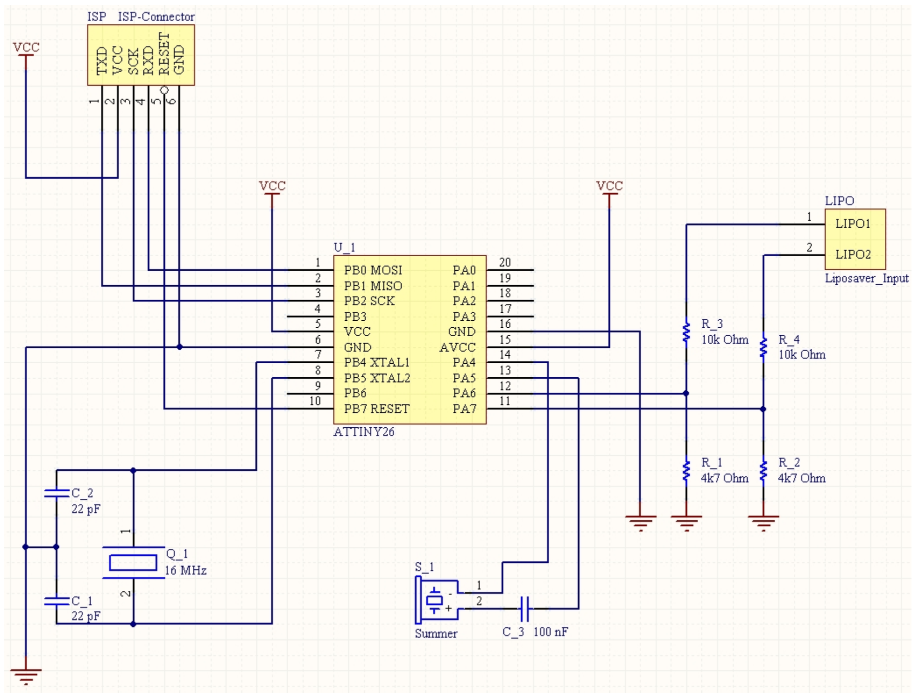


Figure 1.11: Battery protection

Generating a acoustical warning out of the optical warning the LiPo savers have to pass down their signal to the security board. Having a dynamic load on the lithium polymer accumulators (see 2.2), especially in motion of the robot, the voltage can drop temporarily, although the batteries are almost full. This is not a major problem but it causes the LiPo saver to flash shortly. The acoustical warning in contrast should only appear, if the batteries are barely empty. This results in the fact that the LiPo saver signal has to be filtered over time. It was decided to filter the signal over two seconds, to avoid false negative warnings, which were evoked by a shorter time period of the LiPo saver signal. An acoustical warning is only generated, if the signal is constant over this period of time. A piezo speaker, which is driven by a micro controller is generating the acoustical warning. The micro controller is also responsible for the filtering of the signal.

10 A crowbar or crowbar circuit is an electrical circuit used to prevent an over voltage condition of a power supply unit from damaging the circuits attached to the power supply. It operates by putting a short circuit across the voltage source.

The inputs $PA6$ and $PA7$ of the micro controller (see figure 1.11) are connected via a voltage divider to the LiPo savers. The acoustical output is triggered over the outputs $PA4$ and $PA5$ of the micro controller. These outputs are driven inverted to each other to generate a strong signal for the piezo speaker. The pulsing of the circuit is done by a 16MHz crystal oscillator connected to $XTAL1$ and $XTAL2$. The programming of the controller is done via the ISP connector.
The circuit is generating two different kinds of beep sequences depending on which batteries (the servo batteries or the board's batteries) are empty. In a situation where both batteries are empty, the beep sequences are played alternating.

1.5.2 Crowbar Circuit

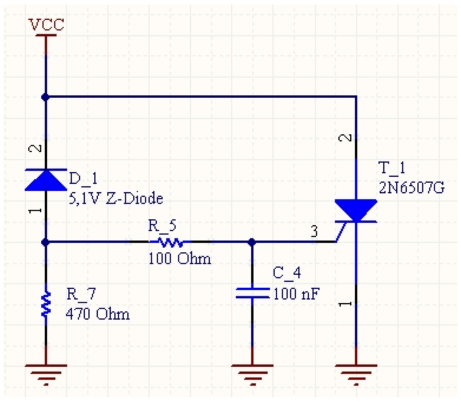


Figure 1.12: Crowbar circuit

The commonly used circuit is working with a thyristor, which gets low resistance in case of an over voltage condition. The circuit creates a bypass in that situation which results necessarily in a blow of one of the fuses. The voltage is adjusted via a Z-diode. In figure 1.12 the diode is the component D_1. It has a value of 5,1V which is adequate for the output voltage for the XTX board (see 1.1). The XTX board is generally working with 5V. For the servo's output voltage the crowbar circuit has to be implemented a second time. Now only the Z-diode is changed to a 7,5V alternative.

1.6 USB-I2C Converter

All the sensors used by the robot are connected to the *carrier board* with help of the I2C bus. The I2C bus is a multi master serial bus which is often used to attach low speed devices. This section will not cover any details about the I2C bus itself or its protocol. This chapter describes the hardware and firmware of the *USB I2C converter*.

1.6.1 Motivation

The *Congatec XTX board* actually has an on board I2C interface which can be used. Unfortunately it turned out that due to bad hardware implementations there is a lot of offset created during the communication. It appeared that even having a clock speed of 400 kHz it was able to transfer only 1 Kbyte/s which is not quite enough considering the variety of sensors it was planning to use. In order to achieve a bigger bandwidth a simpler hardware interface had to be used which is powerful enough to operate the I2C bus at least in fast mode (400 kHz) or even in high speed mode (3,4 MBit/s). After trying different micro controllers the *microchip PIC 18F2550* turned out to be the most applicable. The*18F2550* comes besides many other build in modules with a build in *USB 2.0* interface (full speed and up to 32 end points) and a SSP or I2C (slave/master (interface). A free available C compiler, IDE and USB driver / development environment is also available.

1.6.2 Features

The features implemented in the *USB-I2C converter* are a result of the requirements and wishes made for the converter. Some features like the USB boot loader and 3,3V functionality are a nice bonus resulting out of the selected hardware.

- *USB 2.0* (low speed 1,5 MBit/s and full speed 12 MBit/s)
- I2C full speed (400 kHz) and high speed (1 Mhz)
- USB - boot loader
- Firmware for CDC / MCD device
- USB powered
- Fully functional even on 3,3 V

1.6.3 Realization

Microchip offers a so called USB framework which provides basic USB functionalities. However it is still important that some parameters are adjusted to work properly. This chapter will not cover all the descriptor available in the framework or for a USB device in detail. It is important however to have some basic understanding how USB devices work and are managed, the next section explains are a device needs to be set up.

- Device descriptor
  In general it is true that every USB device has exactly one device descriptor. The device descriptor contains vendor, type, serial numbers and some control information.

- Configuration descriptor
  Depending on how many configurations a device may have a USB device can have one or more configuration descriptors. Information about energy consumption, number and type of interfaces, etc are stored in the configuration descriptors.

- Interface descriptor
  For every interface described in the configuration descriptor needs to be an interface descriptor. Interface descriptors contain information about its type, e.g. HID, and the number of endpoints the interface has.

- Endpoint descriptor
  Every endpoints realised or used by an interface needs to be configured. The Endpoint descriptors take care of that. Endpoints usually are either read or write, only control endpoints can be used as read/write points. Also the size of the endpoints needs to be described.

After the installation of the USB framework an setting up the MPLAB project the settings described above can be applied as needed. After running various tests with different setting configurations the following seems to be the best for this application:

- Boot loader for easy firmware upgrades
- One device descriptor
- One interface with two endpoints (incoming / outgoing)
- Pulling endpoints every 2ms

The framework also created a folder "user" where the actual device code will go. The framework maintains the basic USB functions and has to be executed several times per second. Thus the user code has to execute quite fast, especially it can not contain any "while(true)" loops because it would stall the main framework and the USB device looses the connection to the host.
During the boot sequence the user code initializes the I2C hardware of the*16F2550*. During normal operation the user code reads the USB buffer and analyses it. The protocol used to interface the *USB-I2C converter* will be explained in detail later.

The hardware setup is quite simple since *18F2550* has everything need already embedded. All needed are some resistors, transistors, connectors and some status LEDs. The I2C connectors match form and pin assignment with the I2C connectors used on the carrier board. The USB connector respects the USB pin convention and is assigned as follows:

- Pin 1 - VBUS
- Pin 2 - D-

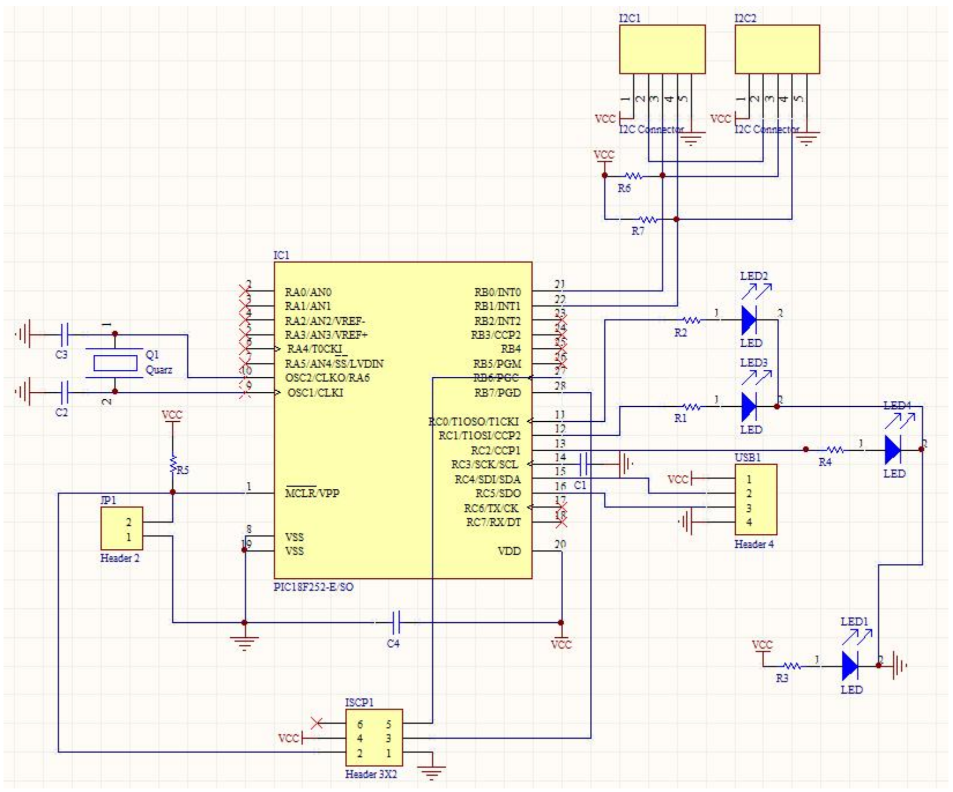


Figure 1.13: Schematic

- Pin 3 - D+
- Pin 4 - Ground

The four status LED indicate:

- LED1 - green - *USB-I2C converter* has power
- LED2 - red - error addressing an I2C device
- LED3 - yellow - error sending data to I2C device
- LED4 - blue - boot loader / scanning for I2C devices

With the *jumper J1* the boot loader can be activated. If*J1* is closed during power on, the *USB-I2C converter* will start the boot loader and a new firmware version can be loaded with help of the windows application "$'USBoot3_5.exe$"'. For normal operation *J1* has to be open.

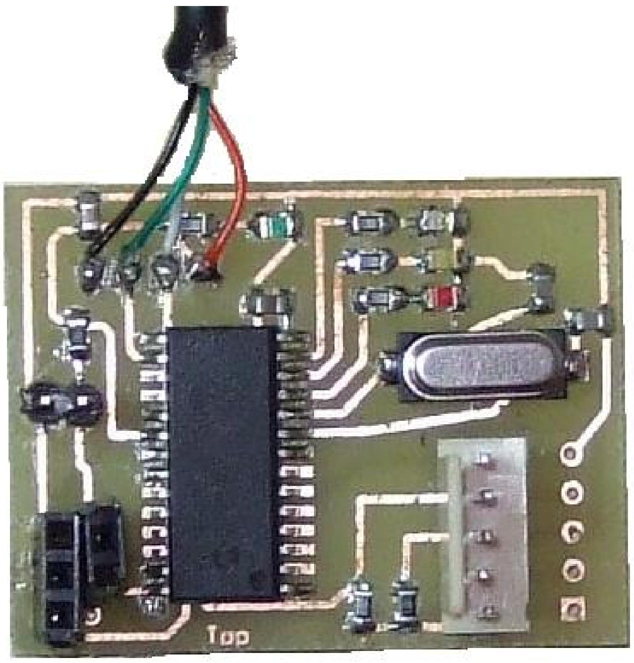


Figure 1.14: USB-I2C converter

1.6.4 Protocol

The first byte always contains the address of the I2C device. To send control commands to the *USB-I2C converter* the address has to be 0. The second byte tells the firmware how many bytes have to be sent or read from the I2C bus. In case of a write operation the following bytes contain the data sent to the I2C bus.

Address (1 Byte)	R/W length (1 Byte)	Data 1 (1 Byte)	...	Data n (1 Byte)

Figure 1.15: Protocol

1.7 Sensors

In this chapter the sensors available on the DohBot's humanoid robot will be introduced. Not all of the presented sensors were used on the *German Open 2007* or at the *RoboCup 2007*.

1.7.1 Gyroscope

The gyroscope is an inertial sensor used to stabilize the robot's motion while e.g. walking. The ADXRS300 gyroscope is an analog device measuring 7 mm x 7 mm x 3 mm (see figure 1.16).

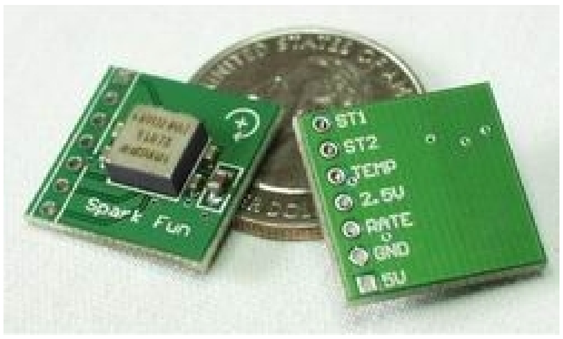


Figure 1.16: ADXRS300 gyroscope

It is a 300 deg/sec angular rate sensor implemented on a single chip including all required electronics. It uses an 80 Hz bandwidth and has a z axis (yaw rate) response. The outputs are analog values ranging from 0 V to 5 V which have to be converted into digital values. For this purpose the carrier board provides an embedded A/D converter which is connected to the I2C bus.

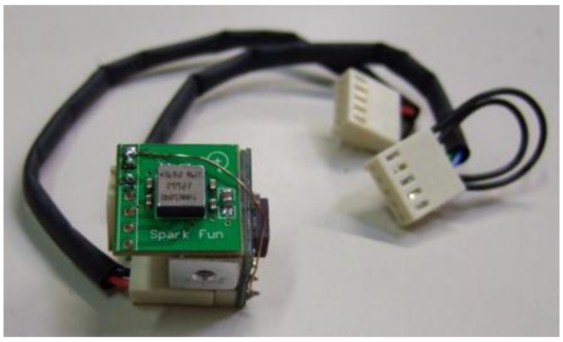


Figure 1.17: Two gyroscopes mounted on a metal cube

To achieve good balancing results two gyroscopes are used so that their rotation axes cover the x-y plane. Two sensors were mounted on a metal cube (see figure 1.17).

Figure 1.18 describes the behaviour of the sensing device when mounted on a robot that is balancing in one direction. The rotation axis is directed out of the package's top. For clockwise rotations it produces values between 2.5 V and 4.75 V and for counterclockwise rotations values between 0.25 V and 2.5 V.

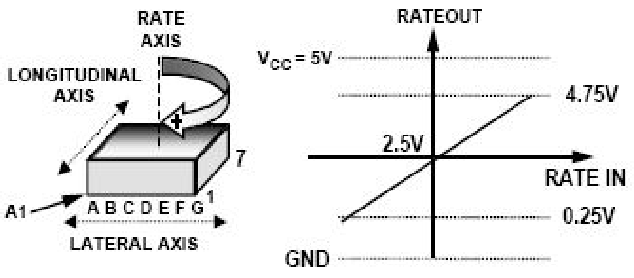


Figure 1.18: Rate out

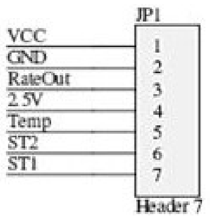


Figure 1.19: Pin header

Pin	Designation	Description
1	VCC	5 V
2	GND	Ground
3	RateOut	Measurement output
4	2.5 V	Reference signal
5	Temp	Temperature
6	ST2	SELF-TEST INPUTS: ST2 RATEOUT response
7	ST1	SELF-TEST INPUTS: ST1 RATEOUT response

The gyroscopes were not used to stabilize the walk in actual robot soccer matches because of a lack of time for developing an accurate model and especially for implementing an interpretation of the sensor's outputs. Doubts that the resolution of the gyroscope is not applicable and that therefore the gyroscope should be replaced by one with a 60 deg/sec resolution were allayed by other teams using exactly the same sensor successfully on the *German Open 2007* and at the *RoboCup 2007*.

1.7.2 Acceleration

The *Bosch SMB 380* is a triaxial low-g acceleration sensor IC with digital output. It allows measurements of acceleration in perpendicular axes as well as absolute temperature measurements. An evaluation circuitry converts the output of a three-channel micromechanical acceleration sensing structure that works according to the differential

capacitance principle.
The *Bosch SMB 380* has a very small package (3 mm x 3 mm x 0.9 mm) which is an advantage and a challenge at the same time. If mounted/soldered directly to the carrier board it consumes only very little space. The problem however is the soldering itself. Unfortunately *Bosch Sensortec* provided an evaluation package with 3.2 mm pinout.

Features

Key features of the *Bosch SMB 380* are the following:

- *USB 2.0* (Triaxial accelerometer)
- Temperature output
- Small QFN package (footprint 3 mm x 3 mm, height 0.9 mm)
- Digital interface SPI (4-wire, 3-wire), I2C, interrupt pin
- Programmable functionality g-range $\pm 2g/\pm 4g/\pm 8g$, bandwidth 25-1500 Hz
- Ultra-low power ASIC with low current consumption and short wake-up time

Internal Filter

The *Bosch SMB 380* has a built-in digital filter bank. The filter bank uses an averaging filter to provide different frequencies. This enables the sensor to be read out at lower frequencies without losing information due to sub sampling. This leads to a signal of higher quality at low read-out rates (see figure 1.20 and 1.21). The frequency can be set to seven levels between 25 Hz and 1500 Hz.

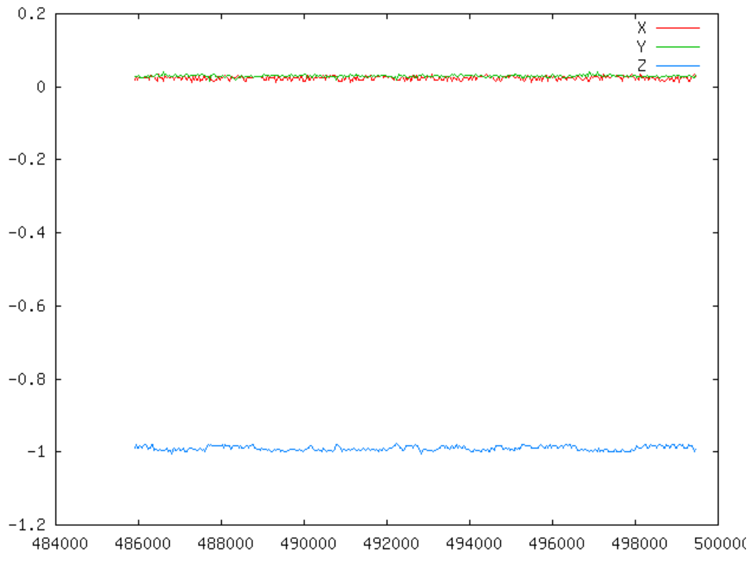


Figure 1.20: *Bosch SMB 380* with filter at 50 Hz and a read-out rate of approximately 25 Hz. Data recorded from a free hanging robot.

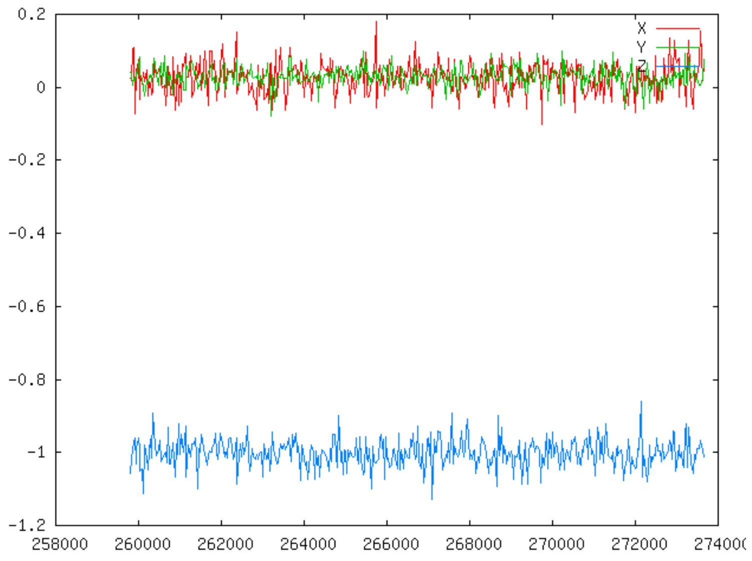


Figure 1.21: *Bosch SMB 380* with filter at 1500 Hz and a read-out rate of approximately 25 Hz. Data recorded from a free hanging robot.

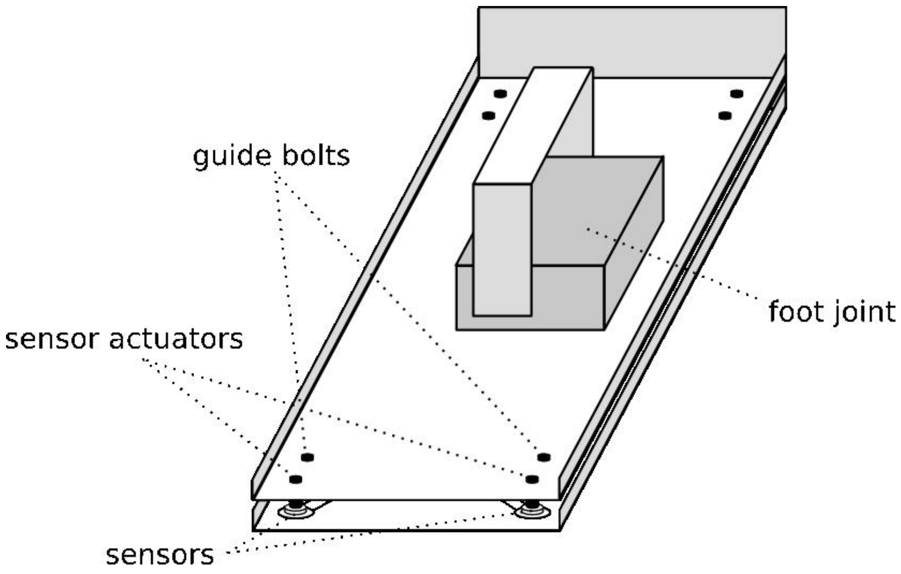


Figure 1.22: Scheme of the robot's foot

1.7.3 Foot sensors

The figure 1.22 shows a drawing of the robot's foot equipped with sensors. The foot sensors measure the force, that the robot applies to the ground in z direction. Each foot is equipped with four sensors, one sensor in every corner of the foot. As sensor elements the *FSR-149* from *IEE* (for the data sheet see [Int]) are used. To read out the sensors data and send it via the I2C bus to the controller board, a micro-controller *At-Tiny26* from *Atmel* (for the data sheet see [Cor]) is used. Figure 1.23 shows that the resistors R_2 to R_9 build voltage dividers with the sensors. Let S_i be the force dependent resistance of the sensor i. Then the voltage U_i measured by the A/D converter of the micro-controller is:

$$U_i = \frac{R}{R + S_i} U_0$$

According to the data sheet of the sensors the sensor's resistance is approximately reciprocally proportional to the force.

$$S_i = \frac{c_i'}{F_i}$$

So:

$$F_i = c_i \cdot \frac{U_i}{U_0 - U_i} \tag{1.3}$$

A way to calibrate the sensor is to measure the parameter c_i for a known calibration force, and calculate all other forces using equation 1.3. This kind of calibration turned out to be insufficient because it is only correct for the calibration force. Another way to calibrate the sensors is of course to measure the voltage for a greater number of forces, and linearly

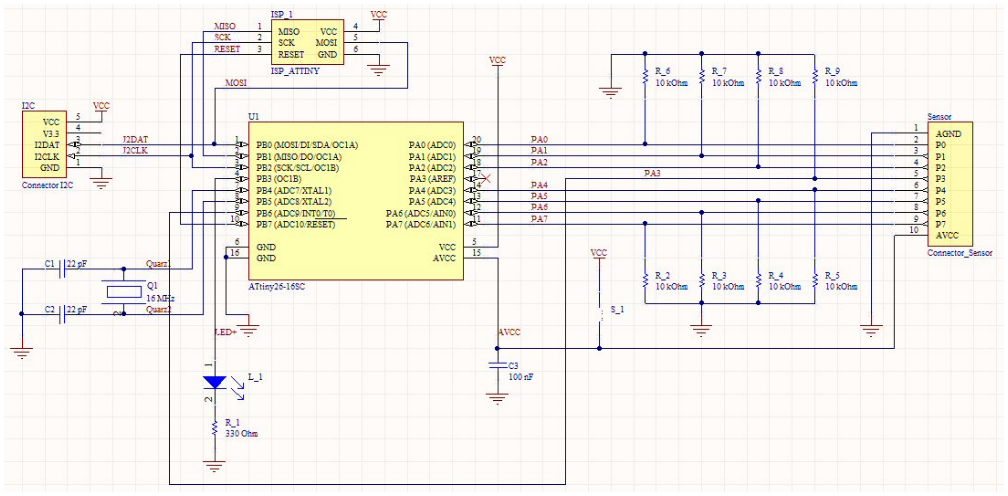


Figure 1.23: Scheme of the foot sensor controller board

interpolate the forces in between this measuring points. The problem with this method is that it systematically overestimates the force. The solution to this problem is to combine the two ways of calibration. Instead of only measuring the voltages of the given calibration forces, the parameter c_i in equation 1.3 is additionally calculated. Between this measuring points only the parameter $c(U_i)$ has to be interpolated linearly and is then used to calculate the appropriate force.

2 Hardware

The robot's mechanics plays a very important role in the whole process of robot development. It affects the way the robot walks, kicks or in general moves. It has to be robust and lightweight. In order to satisfy all requirements a new robot was developed instead of using an industrial manufactured configuration.

2.1 Servos

Servos belong to the most important parts of the robot because they are responsible for stable and precise movements. In the current configuration the robot possesses 18 servos, 6 of them for the arms (3 per side) and 12 for the legs (6 per side).
On the one hand the servos must be strong enough to allow reliable movements and on the other hand the maximal energy consumption is limited.
The first revision of the robot used several types of servos (e.g. *Hitec HSR-5995TG*, *Hitec HS-5995TG*, *Diamond DS9500*, *Hitec HS-5945MG*). Depending on their qualities different servo types were used in different sections of the robot: arms, hip and feet. This was not practical because every supplement had to be bought for each servo type separately. Another problem was that some servos were not strong enough for specific movements (e.g. stand up). Therefore the next revision of the robot was build with servos of one type (*Hitec HSR-5990TG*).
The biggest problem of this servo is its high heat development. Most of the heat is produced by electronic circuits which can be cooled passively or actively. The usage of heat sinks provides good heat protection for servos which do not have too much load (e.g. arms), but for servos which are permanently heavily loaded (e.g. knee joint by walking), passive cooling is not sufficient. Some of the teams participating in the *RoboCup* used therefore active cooling based on small fans attached to their servos. This type of cooling provides very good heat dissipation, but the power consumption raises.
In the current configuration, the servos are cooled passively, but the cover of the servo's circuits is shifted a few millimeters outwards in order to improve the air flow. Additionally, cooling spray is used for the heavily loaded joints. A disadvantage of applying the cooling spray is that the robot has to be taken off the game. Anyway, it is necessary to prevent the servos from being damaged.
Another problem were plastic gear wheels in the transmission (see figure 2.1) which got broken and blocked the servo when they were exposed to heavy load and high temperature. The problem was solved by replacing the plastic gear wheels with more robust metal ones.

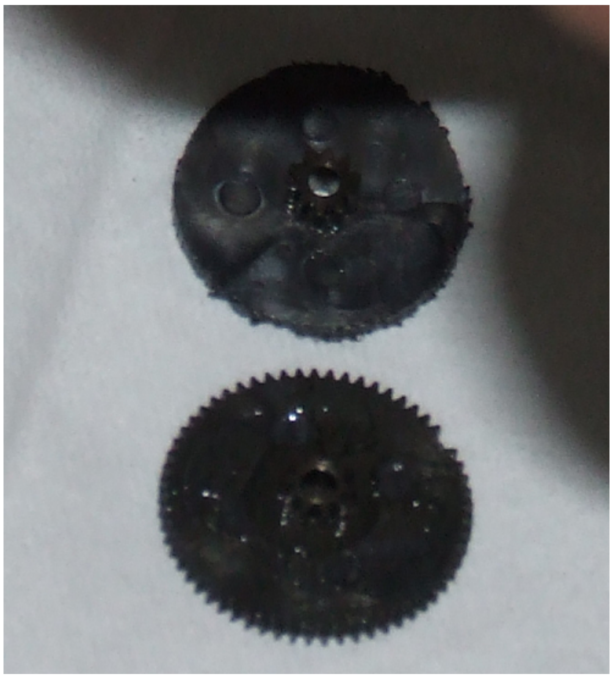


Figure 2.1: Damaged gearwheel

2.2 Accumulators

To operate the robot an adequate power supply is needed. The power supply has to be portable which means small and lightweight. Its capacity has to be sufficient for at least 10 minutes of nonstop operation which is one half of an robot soccer game. The current configuration includes two sets of Lithium-Polymer (LiPo) accumulators with 1100 mAh and 11.2 V. Each set consists of two accumulators. One set supplies the servos and the other one the board.

To protect the accumulators against deep discharge (which can cause the accumulator to explode) a LiPo saver is applied (see chapter 1.5.1) for each set of accumulators. The LiPo saver provides a possibility to control the accumulators charge state and enables the user to change them, if necessary.

For the board one set of fully loaded accumulators provides enough power for the whole game which means 20 minutes of continuous operation.

The servos, although, need more power than the board. In the best case the servo's accumulators have to be changed only once a game. But if the servos are heavily loaded e.g. by many stand up movements both accumulators are empty within several minutes. This is the normal case. A possible solution would be reducing the weight of the robot's parts or to make movements more energy efficient.

2.3 Upper Body Design

The robot's upper body is mainly divided into two parts - hip and torso. Both had to be customly designed, due to the special needs of the customized PCB layout, the arms' and legs' configuration and the camera's position.
The first revision was produced from bended aluminum sheets and was used until the *German Open*. Because of the experiences made during the tournament, i.e. discovering problems with the body's structural integrity, the design and the production process were altered. Since the second revision, the design is made with a CAD-Tool and the body is produced from solid aluminum.

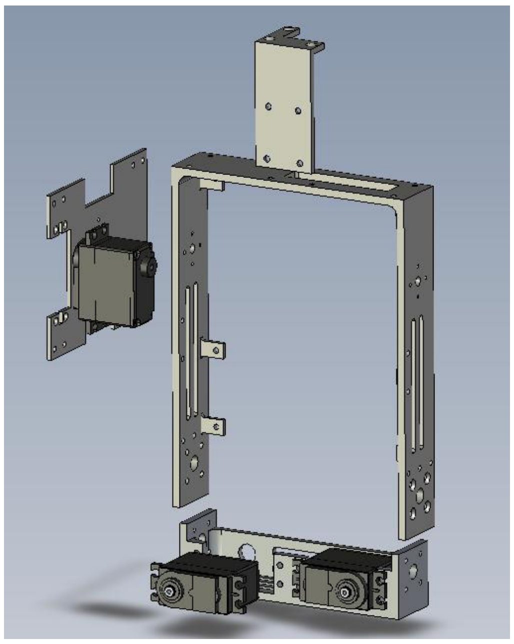


Figure 2.2: Upper body (exploded view)

2.3.1 Hip

A good hip design is pretty important for the robot, as it has a lot of side effects. The leg design, the integrity of the torso, and - as experienced especially during the *German Open 2007* - also the walking stability is affected by the hip design.

Revision 1

As there has been the decision to integrate only the two servos for spreading legs into the hip, it was looked for an easy solution to keep the design simple but functional.

This has been achieved by customizing a *Multipurpose Dual Bracket (ASB - 15)* from *Lynxmotion*[11].
The choice of material was aluminum sheets which is very lighweighted on one hand, but hard enough to avoid easy deformation or breaking on the other hand. By planing the bended edges carefully, the structure gets stiff enough to compensate the occuring stresses.

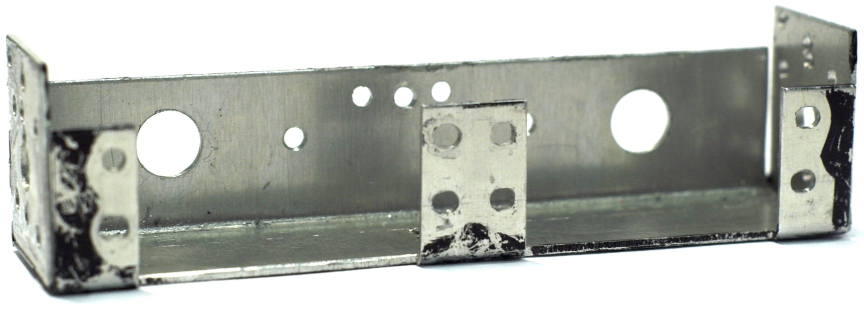


Figure 2.3: Old Hip Design

This material has been experienced not to be as solid as it was thought; and there was also the disadvantage that bended foils can not form closed structures. For this reason the bended edges got soft and instable, and so slightly deformed the hip while weakening the material at the same time. Because of this there have been massive problems in walking and additional work on calibration.
This enforced a redesign of the hip after returning from the *German Open 2007*.

Revision 2

As there were mainly problems caused by the material and the way of production, but not with the basic design itself, further work was concentrated on those problems.
To solve this it has been decided to produce the hip from solid aluminum by milling, so that a closed structure with nor weak nor open edges can be obtained. The production technique itself allows to work very precisely now allowing to precisely embed the servos into the hip which also stablizes their bearings.
Regarding a servo's great heat dissipation and to gain lesser weight, the new hip has been provided with some notches, taking care not to infringe the structural integrity.

The advantages of this new design are considerably more structural integrity, improvements of the walk and its stability, increased and more precise reproducability and a

11 www.lynxmotion.com

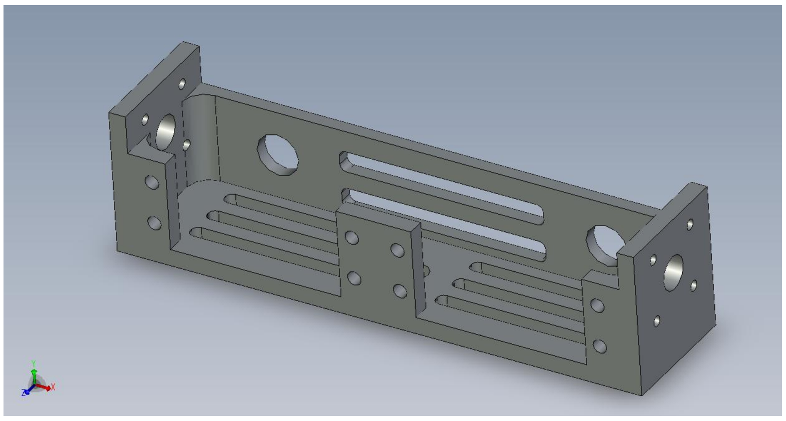


Figure 2.4: New Hip Design

simpler and faster assembly of the robot.

2.3.2 Torso

The most problematic tasks on the design of the torso were the size, weight and shape of the PCB stack. But the arm design, the mounting of the camera and the dead weight had to be considered as well.
Trying to keep the design simple it was decided to use an U-profiled torso which is connected to the hip with screws, to ensure separate and easier maintainance of both parts. The arms, the camera and the mountings for the PCB stack and a safety cage are screwed to that basic frame.

Revision 1

The first revision was - like the hip - made from bended aluminum sheets, which lead to the same structural problems. But those were even worse here as the torso was highly deformed by the forces through the robot's fall-downs and the forces on the arms during stand-up moves.
In this revision also two special brackets had to be mounted to the torso for installing the PCB stack. These were not just hard to produce because of the PCB layout, but also took a lot of space and had to be additionaly insulated to avoid short-circuit damages to the PCB stack. Another problem was the tricky and time-consuming installation of the stack and the brackets themselves.

Revision 2

Facing severe problems with structural integrity and difficult assembly to the robot, it was likewise decided to mill the torso out of solid aluminum and to produce an optimized new design.
As the problems with integrity should be solved by the new material, design work was concentrated on more functionality and easier assembly. For these reasons the mountings for the PCB stack have been integrated to the torso instead of using additional brackets, which provides more space and easier installation and solves the danger of short-circuit damages by limiting the area of contact to uncritical regions. Also the provisions for installing the camera, the arms, the safety cage and wires as well as the connection to the hip have been precisely integrated into the design.
This additionaly makes the new torso much easier and faster to assemble and to produce.

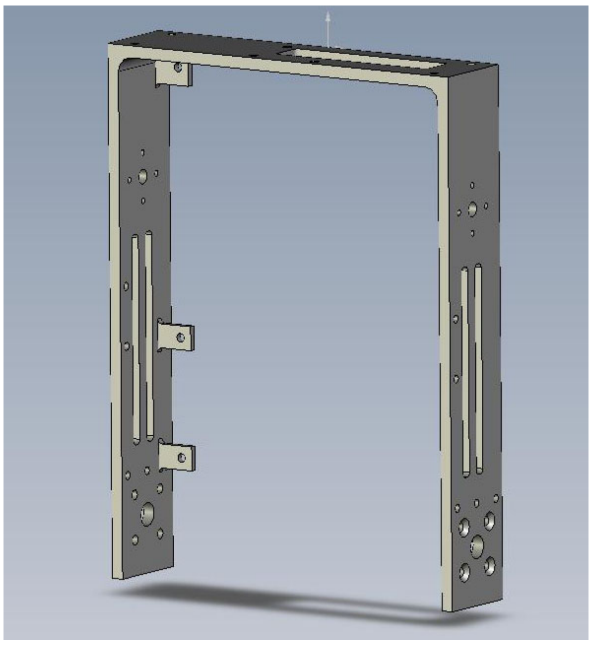


Figure 2.5: New Torso Design

From the torso's second revision's design a lot more structural integrity and reliability for the robot has been experienced during *Robocup 2007*.

2.4 Battery Holders

In the first version of the robot, two Lithium-Polymer accumulators were mounted on each foot, one on the front and one on the back end. They were attached to the feet by simple Velcro fasteners. Allthough they did not got loose during operation, they

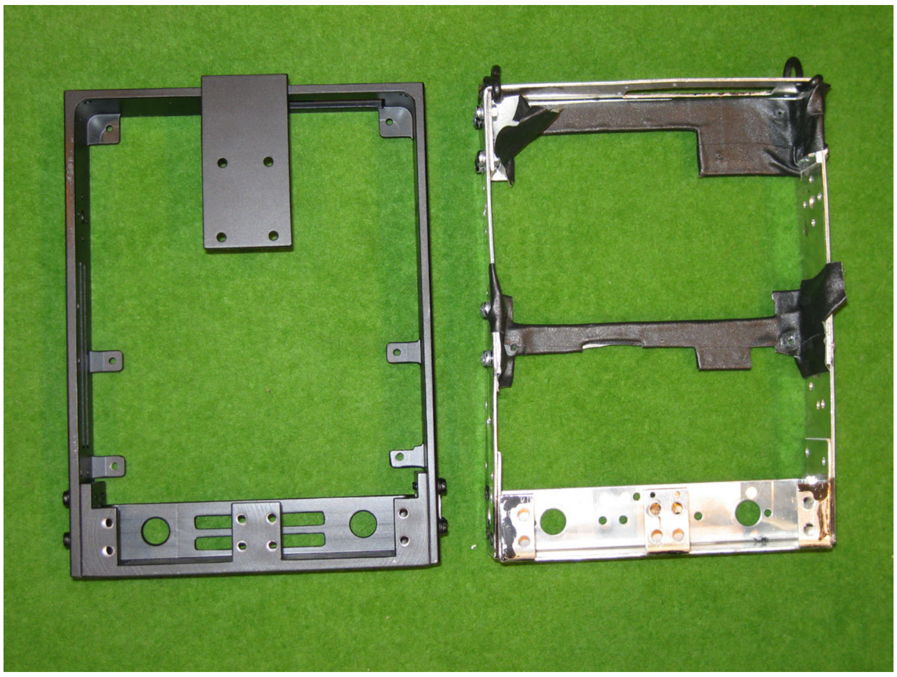


Figure 2.6: Comparison Torso Design

have not been totally fixed to their positions and there was an immanent danger of an opponent robot kicking away a battery.
So after the *German Open 2007* there has been a need for a better mounting for the batteries. Another reason were the facts that the walk was thought to be stabilized by having just one battery per foot and the newly tested balancing engine also needed more weight at the hands for its efficiency.
For those reasons a cage has been designed for the batteries made out of aluminum sheet, which is lightweight and extensively open to dissipate any heat from the batteries. It is also universally mountable to the robot at the same time. As the holders are customized for the batteries, they now have a tight fit, but are also pretty easy to exchange. The battery's fuse is now used for fastening the battery in the battery holders.
During *Robocup 2007* good experiences have been made with the new battery holders, even if some little improvements, like widening the notches for the fuses a bit, could be made to make the exchange of batteries during matches faster and easier.

2.5 Foot Design

The foot design does not seem to be so important compared to other works done for the robot. In fact, this is only partly true. In the beginning, the feet delivered with the initial setup where used.

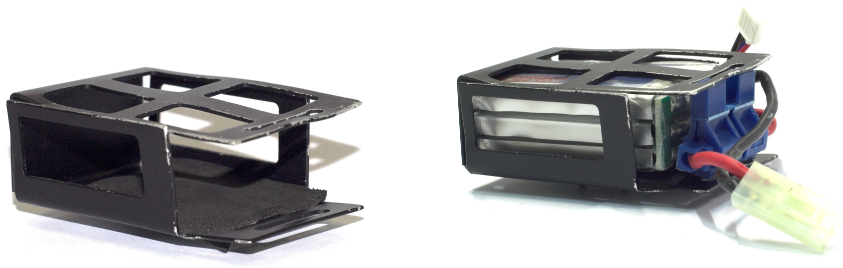

(a) Design (b) LiPo Installed

Figure 2.7: Battery Holder

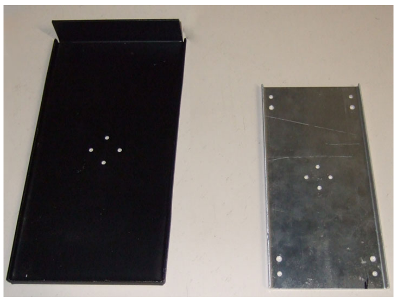

(a) Foot Design

Figure 2.8: Comparision of new and old foot design

The dimensions of these feet were remarkably smaller than allowed by the official rules. As a result of that the whole robot was somewhat unstable during any movement. Especially when it came to dynamic stability like in kicking movements, it was almost impossible to keep the robot on its feet.
Since these feet where the only ones which could be bought for that robot, it was decided to develop an own foot design.
Feet were designed which were close to the maximum allowed by the rules. The actual design of the feet did not change that much, but this was not necessary since the only demand was to have bigger feet, basically.
The choice of material was aluminum sheets which is very lighweighted on one hand, but hard enough to avoid easy deformation or breaking on the other hand. By planing the bended edges carefully, the structure gets stiff enough to compensate the occuring stresses.

This material has been experienced not to be as solid as it was thought; and there was also the disadvantage that bended foils can not form closed structures.

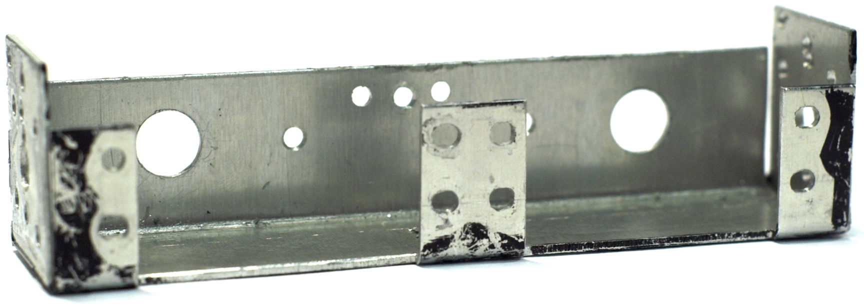
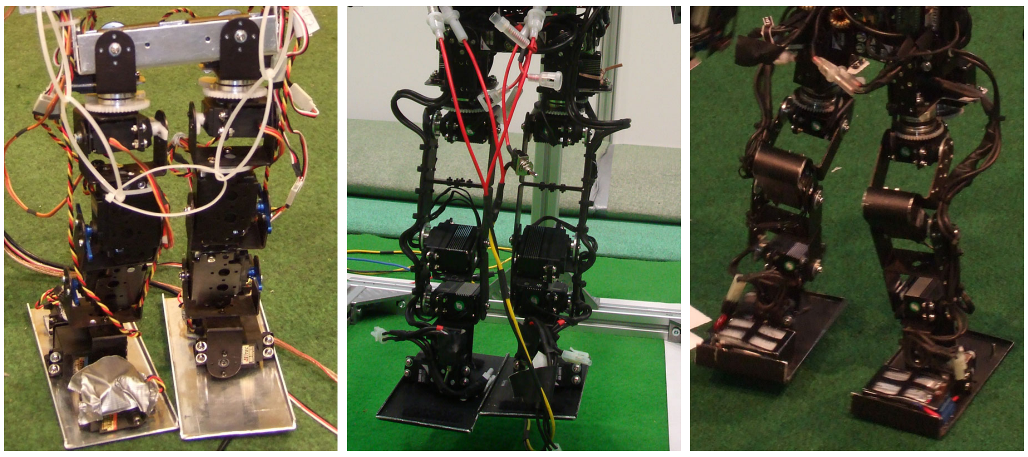


Figure 2.9: Old Hip Design

2.6 Leg Design

A good leg design is one of the most important things to have besides solid software. In fact, it is so important that it was changed three times during the work on the robot. Due to the lack of knowledge in humanoid soccer there was not paid much attention to the leg design at the beginning. Thus, the initial leg design of the robot was used for the first couple of weeks.
After first tests with the robot, the knees were moved a little bit higher, as an improvement on walking and kicking was expected by that.



(a) First Re-Design (b) Intermediate Design (c) Final Design

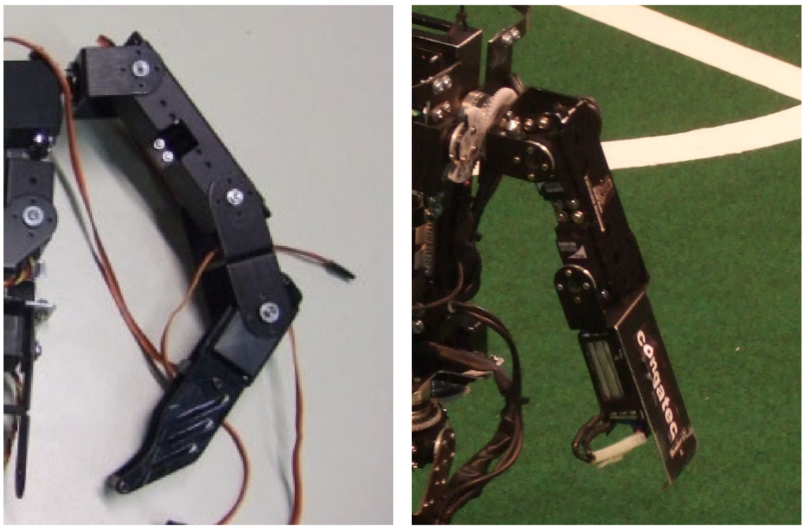
Figure 2.10: Leg Design (2)

After some time the leg design was changed again, since the robot was still pretty tiny and it should get as close to the maximum height allowed by the rules as possible. The main reason for this were the dimensions of the upper body. The maximum width allowed is relative to the height of the robot. So the robot had to be pretty tall to meet

those regulations. This time, the small C-brackets of the legs were replaced by longer ones. This way the robot gained about 10 cm in total height.
This leg design was used on the German Open 2007. It did not work out as well as expected. Due to limitations in the knee servos of the robot the whole walk did not function as intended.
Pretty soon after the tournament, the design of the legs was changed once again. This time the long C-brackets were kept, but the actual design was changed. The knee servos were slightly shifted and rotated, so that it was possible to move them at least as far as needed by the walking engine.
With this leg configuration the robot was participating in RoboCup 2007 in Atlanta. This design was working a lot better than the prior designs. And still - it will be not kept in future. The proportion of the legs compared to the remaining body is too high. This makes the walk and the general moving of the robot unnecessarily hard to control. The next step will be to keep the design of the legs and change some of the long C-brackets back to short C-brackets. This will also help getting the center of mass closer to the ground - which by itself already helps to stabilize the robot.

2.7 Arm Design

During the work on the robot, the arm design was changed twice. The first time it was necessary because the space inside of the robot was needed for the PCB stack and the initial arm design blocked this space.



(a) Initial Arm Design (b) Final Arm Design

Figure 2.11: Arm Designs

The change was part of the whole redesign of the upper body for being able to carry the PCB stack. The result can be seen in the picture 2.11.

During this reconstruction, neither the carrier board nor the congatec board was available only a blueprint of the carrier board's dimensions was available at that time. The new arm design uses parts of the old legs. The long bracket in the upper part of the arms used to be the shinbone of the robot.
Nevertheless, the design had to be changed once more to make the arm construction more stable. Inspired by the leg design, it was used as a draft for the redesign. The rotation parts of the legs were also used for the arms. This way the arms got a lot more stable and now it is even possible to lift the whole robot (4kg) by grabbing only one arm.

3 Operating System

3.1 General

One of the early tasks was to decide which operating system should be used by the robot. The choices were obviously some sort of tripped down or embedded *Windows*, or a small *Linux* distribution. Since a lot of work on the hardware side was expected and the majority of the group was more fluent with *Windows*, it was decided to give *Windows XP Embedded* a try. All drivers needed for the *Congatec XTX* Board, wireless stick and cameras are available for *Windows*. Now all drivers are also available for *Linux* so that experiments with *Linux* might also be very interesting. This chapter gives a little introduction to *Windows XP Embedded* (XPE) and will give an overview about the configuration used. It will also list the components that where developed during the last year.

Windows XP Embedded is basically a regular *Windows XP* except that it can be configured very individually. The footprint ranges from a minimum of about 20 MB up to several hundred megabytes. Usually no special hardware drivers are needed or the *Windows XP/2000* drivers can be used. XPE is, since Service Pack 2, able to start from a USB drive, such as a memory stick. To protect the flash memory of the stick from unnecessary write cycles, a write filter can be used.
The write filter puts a logical layer (see figure 3.3) between the data system and the harddrive. Write requests are redirected to the logical layer and not directly written to disk. XPE offers two write filters, the *Advanced* and the *File Based Write Filter*. The *File Based Write Filer* (FBWF) was introduced with *SP2* as well. Its main advantage over the *advanced write filter* is, that it can be applied for a whole disk, partition or just certain files. It turned out to be useful to have the whole drive protected except the working folder "writable" and the registry. Since everything, except the working folder and the registry, where write protected it was on the one hand ensured that the flash disk did not get too many unnecessary write cycles and on the other that the operating system could not get destroyed by accidentally deleting some files or improper shutdowns.

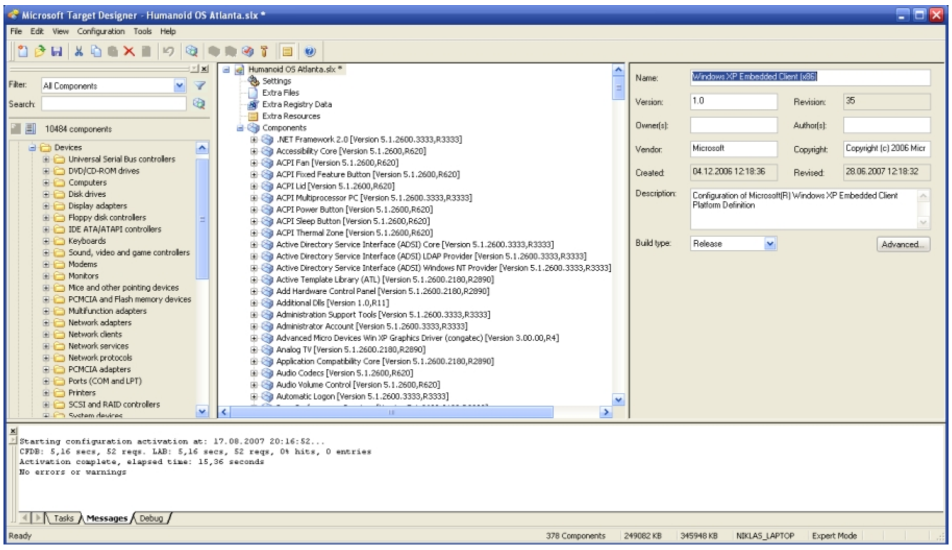
To manage the FBWF easier and more convenient, a tool was developed to perform some basic operations. It will be introduced later.

To configure an XPE, various tools were developed by *Microsoft*. A database keeps track about all the components available, called the *Windows* XPE repositories. The *Component Designer* needs to be used if new components are being developed. With help of the *Component Designer* the actual setup is created. After all components are collected and all dependencies are resolved, an image of the configuration can be build.

During the first system boot the *first boot agent* configures the system and registers the components.

3.2 Configuration

The configuration or the composition of the components used for the XPE image is made with the *Target Designer* shown in figure 3.1. On the left hand side the basic categories like networking, systemtool, hardware components, etc are listed. The actual configuration is shown in the middle of the screen, on the right hand side the basic properties of the components can be set. The *administrator account* component for example offers the option to specify an administrator password. Some components like the *Explorer Shell* have dependencies with other components, usually the *Target Designer* automatically adds the dependent components. It happened however that after the installation and even all dependencies where resolved some functions or property dialogs had no effect and it appeared that only the GUI itself was added. Apparently there is no automated mechanism provided to find the missing components and it turned out to be very hard to find the missing components manually.



Filter: All Components
Search:
10484 components

- Devices
  - Universal Serial Bus controllers
  - DVD/CD-ROM drives
  - Computers
  - Disk drives
  - Display adapters
  - Floppy disk controllers
  - IDE ATA/ATAPI controllers
  - Keyboards
  - Sound, video and game controllers
  - Modems
  - Monitors
  - Mice and other pointing devices
  - PCMCIA and Flash memory devices
  - Multifunction adapters
  - Network adapters
  - Network clients
  - Network services
  - Network protocols
  - PCMCIA adapters
  - Ports (COM and LPT)
  - Printers
  - SCSI and RAID controllers
  - System devices

- Humanoid OS Atlanta.slx *
  - Settings
  - Extra Files
  - Extra Registry Data
  - Extra Resources
  - Components
    - .NET Framework 2.0 [Version 5.1.2600.3333,R3333]
    - Accessibility Core [Version 5.1.2600,R620]
    - ACPI Fan [Version 5.1.2600,R620]
    - ACPI Fixed Feature Button [Version 5.1.2600,R620]
    - ACPI Lid [Version 5.1.2600,R620]
    - ACPI Multiprocessor PC [Version 5.1.2600.3333,R3333]
    - ACPI Power Button [Version 5.1.2600,R620]
    - ACPI Sleep Button [Version 5.1.2600,R620]
    - ACPI Thermal Zone [Version 5.1.2600,R620]
    - Active Directory Service Interface (ADSI) Core [Version 5.1.2600.3333,R3333]
    - Active Directory Service Interface (ADSI) LDAP Provider [Version 5.1.2600.3333,R3333]
    - Active Directory Service Interface (ADSI) Windows NT Provider [Version 5.1.2600.3333,R3333]
    - Active Template Library (ATL) [Version 5.1.2600.2180,R2890]
    - Add Hardware Control Panel [Version 5.1.2600.2180,R2890]
    - Additional Dlls [Version 1.0,R11]
    - Administration Support Tools [Version 5.1.2600.3333,R3333]
    - Administrator Account [Version 5.1.2600.3333,R3333]
    - Advanced Micro Devices Win XP Graphics Driver (congatec) [Version 3.00.00,R4]
    - Analog TV [Version 5.1.2600.2180,R2890]
    - Application Compatibility Core [Version 5.1.2600.2180,R2890]
    - Audio Codecs [Version 5.1.2600,R620]
    - Audio Volume Control [Version 5.1.2600,R620]
    - Automatic Logon [Version 5.1.2600.3333,R3333]

Name:	Windows XP Embedded Client (x86)		
Version:	1.0	Revision:	35
Owner(s):		Author(s):	
Vendor:	Microsoft	Copyright:	Copyright (c) 2006 Micr
Created:	04.12.2006 12:18:36	Revised:	28.06.2007 12:18:32
Description:	Configuration of Microsoft(R) Windows XP Embedded Client Platform Definition		
Build type:	Release		

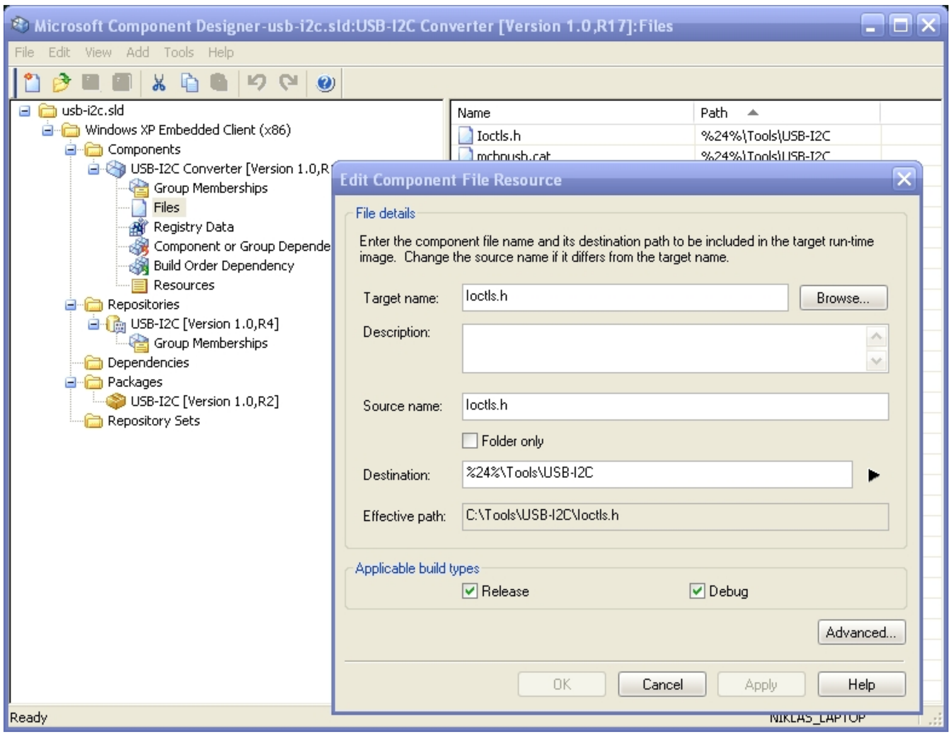
```
Starting configuration activation at: 17.08.2007 20:16:52...
CFDB: 5,16 secs, 52 reqs. LAB: 5,16 secs, 52 reqs, 0% hits, 0 entries
Activation complete, elapsed time: 15,36 seconds
No errors or warnings
```

Ready   378 Components   249082 KB   345948 KB   NIKLAS_LAPTOP   Expert Mode

Figure 3.1: Target Designer

3.3 Components / Drivers

The use of components makes the installation process very easy. Components usually contain all the information needed to e.g. integrate drivers for a wireless networks card in the system. A component contains files, registry settings and usually allows predefining some settings. Once a component is created and integrated in the image there is no need to install or configure any hardware after the first boot of the system. To manage, create and edit components the *Component Designer* shown in figure 3.2 is used.


Microsoft Component Designer-usb-i2c.sld:USB-I2C Converter [Version 1.0,R17]:Files

Edit Component File Resource

File details

Enter the component file name and its destination path to be included in the target run-time image. Change the source name if it differs from the target name.

Target name: Ioctls.h

Description:

Source name: Ioctls.h

Folder only: ☐

Destination: %24%\Tools\USB-I2C

Effective path: C:\Tools\USB-I2C\Ioctls.h

Applicable build types

Release: ✓

Debug: ✓

Figure 3.2: Component Designer

The current XPE image counts 378 components and is about 333 MB in size. The following list gives an overview of the most important components used:

- ***Congatec x945***
- ***USB boot 2.0***
- ***File Based Write Filter***
- ***Networking components***

In addition to the predefined components it was also necessary to develop own components to allow an easy and convenient integration of the files, tools and needed drivers.

- ***Additional libraries and files***
  This component contains some additional files and programs that might come in handy, e.g. an text editor. It also contains a collection of files that did not belong to a specific driver or program.

- ***Creative Live! Cam Notebook Pro***
  This package contains the drivers and software for the notebook webcam that is located at the lower torso of the robot.

- ***D-Link wireless lan***
  The D-Link wlan component contains only the drivers for the wireless USB-Stick. This component could be improved by adding some networking tools.

- ***Omnivision camera***
  This component contains only the drivers for the omnivision camera, the tools for testing and debugging are in a separate package.

- ***Omnivision camera tools***
  This package consists of the tools for testing and debugging. Because it is dependent on the *omnivision camera* component, it cannot be used by itself.

- ***USB-I2C converter***
  Contains the USB-drivers for the USB-I2C converter - that are in particular drivers for the converter itself and also for the bootloader. The bootloader tool and a test tool are also included.

3.4 File Based Write Filter

As it was decided to use Windows XP Embedded (XPE) as the operating system, some adjustments had to be done to it.
The main adjustment was the implementation of the *File-Based Write Filter* (FBWF), that allows XPE to maintain the appearance of read and write access to write sensitive or read only storage. FBWF makes read and write access transparent to applications. Basically, it enables enhanced flash memory reliability through stateless operation, reduced wear, and improved servicing functions.

3.4.1 FBWF Installation and Configuration

The FBWF has been installed offline during the design phase using the *Target Designer*. Once installed, it can be configured using a command shell. In order to make configuration easier and faster, a little tool was developed to take care about the basic settings. The FBWF comes with the FBWF API which gives direct access to all the functions provided by FBWF. Until now, only a few functions are implemented, since only very basic functions were needed so far. Extending the tool, if needed, is very easy though.

3.4.2 FBWF In Detail

The FBWF introduces several new features not found in EWF *(Enhanced Write Filter)*: greater write filter transparency to applications, selective write through, commit and restore, improved overlay memory utilization and finally an enhanced API. Like the Enhanced Write Filter, FBWF prevents writes to one or more protected volumes and caches all writes to protected volumes in an overlay cache on a volatile store. But because FBWF operates at the file level rather than the sector level, it provides several features not found in EWF that will be presented in the following section.

3.4.3 FBWF Features

- ***File and Directory Management Transparency***
  Describes file and directory management functions available to applications.
- ***Selective Write Through***
  Describes how to allow writes to specified underlying protected volumes.
- ***Selective Commits and Restores***
  Describes how to commit writes from overlay cache to underlying protected volumes and how to discard changes in overlay cache and restore the view to the underlying protected volumes.
- ***FBWF Memory Optimization***
  Describes how FBWF optimizes utilization of overlay cache.
- ***FBWF API***
  Describes the FBWF API set.

3.4.4 The FBWF Tool

FBWF-specific functions used by the tool (used to manage the system-wide write cache):

- **FbwfEnableFilter**
  Enables write filtering in the next session.
- **FbwfDisableFilter**
  Disables write filtering for the next session.
- **FbwfProtectVolume**
  Enables write protection for a specified volume.
- **FbwfUnprotectVolume**
  Removes write protection for the specified volume. These functions are used for file commit and restore.
- **FbwfCommitFile**
  Writes the cached file overlay to the physical disk file.

Other functionality:

- **ShutdownWindows**
  Shuts down all open applications and the carrier board.
- **RebootWindows**
  Reboots the carrier board.

FBWF operates at the file level to redirect all write requests targeted for protected volumes to overlay cache. The figure 3.3 illustrates the relationship of FBWF to file system I/O and Windows sub-system components.

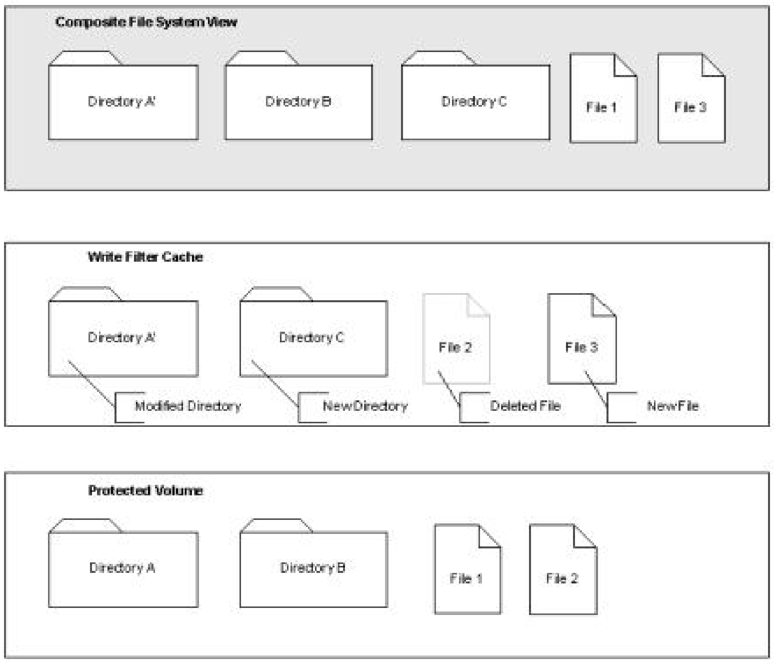


Figure 3.3: Layer view of the FBWF

Figure 3.3 composite view presented by FBWF shows how modified files, File2 and File3 are represented in the cache but not touched in the underlying volume.

4 Software

4.1 Simulator

The simulator is a very important tool in the development process of a robot. It can be used for many steps in the process - beginning from the lowest level of debugging the walking engine, through making the first sketches of the walk or special moves, up to the development of behaviours. The last step could be said to be the main purpose since there is really much testing needed and the time for testing the behaviour on a real robot is mostly very limited. Besides this rebooting the robot for testing small changes is very time consuming.

4.1.1 Previous simulator

Until now, *SimRobot* was used for the simulations. It is a 3D simulator developed at the University of Bremen, that allows the simulation of arbitary robots. The *Microsoft Hellhounds* used *SimRobot* already for the *AIBO* simulations and the *BreDoBrothers* adapted it then for the humanoid robot.
Its key features are:

- *ODE*[12] physics engine.
- Sophisticated scene-management with a XML-based scene description.
- Quite large set of simulated sensors and actuators.

Unfortunately, the physical simulation was not satisfying for a humanoid robot simulation and the overall performance was also a major argument for the decision to take a look at other simulators.

4.1.2 Requirements for the new simulator

Since the simulator was supposed to be used for the next years, the choice had to be well-grounded, so the requirements were elaborated very thorough.

Major requirements:

- **Performance**
  For a comfortable work, the simulation should run very smoothly at about 30 fps. In the best case with multiple robots.

12 Open Dynamics Engine www.ode.org

- **Best possible physical simulation**
  This is a really important issue, because the sketches of the walking engine or animation moves should be made and tested in the simulator.
- **Decoupling from the robotcode**
  The robotcode has to be connected by a TCP/IP network connection.
- **Ability to simulate different kinds of sensors**
  Omnivision camera, gyroscope etc.
- **Stability**

Minor requirements:

- **Ability to run a simulation with multiple robots**
- **Easy to maintain**
- **Good documentation**

4.1.3 The candidates and their features

After a couple of weeks of searching and testing, two alternatives were extracted as the potential candidates:

- **USARSim**
  This simulator is mainly used in the rescue league. Based on the commercial *UnrealEngine* and equipped with *UnrealEd*, a powerful editor, it seemed to be an interesting option. The robot-to-code connection is however made through a workaround using the *Gamebots* mod for *Unreal* - so it was not an optimal solution. Although the biggest argument against it was the closed-source engine, which induces unnecessary restrictions. Overall, this simulation environment did not give an impression to be comfortable, consistent (third-party tools needed) and expandable.
- **Microsoft Robotics Studio (MSRS)**
  It is a quite new simulation environment. The main features are:
  - Service oriented architecture
  - Managed and very fast graphics engine (based on XNA[13] - a set of game development tools)
  - Wrapped Ageia PhysX engine (supports hardware acceleration)
  - Simplified concurrency programming library (CCR)

13 XNA's Not Acronymed

An additional advantage is the fast growing community, where the MSRS developers also participate in.

Finally, the choice fell on Microsoft Robotics Studio - not only because of its features, but also due to the fact, that it is permanently developed further.

4.1.4 MSRS Simulator - architecture overview

The simulation architecture of the simulator consists mainly of two parts: the simulation environment and the robot entity (as shown in figure 4.1). The third, dashed part is the external robotcode.

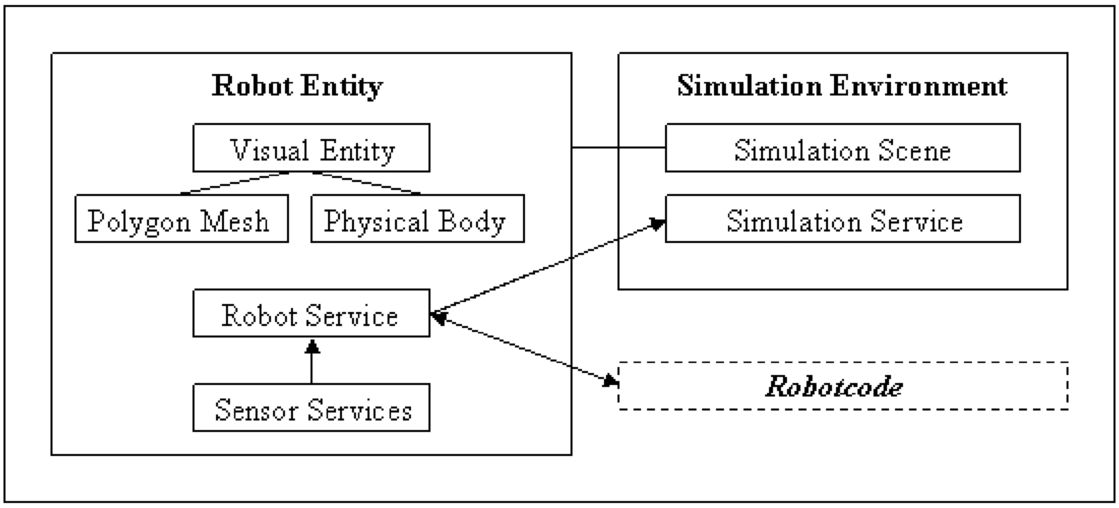


Figure 4.1: Simulator architecture

All the parts will be elaborated in detail in the following sections, but for a better understanding of the whole process, it is also important to introduce the order of events, that are necessary to establish a simulation. The following steps are needed:

1. Start the simulation environment service and set up the scene
2. Create the robot entity, its service and connect them
3. Establish a connection between the robot service and the external robotcode
4. Start the main simulation loop (connection, sensor and movement management)

4.1.5 Simulation Environment

The simulation environment is responsible for the macro-management (i.e. interaction) of all simulated entities. It is built up from two parts: the simulated scene (see figure

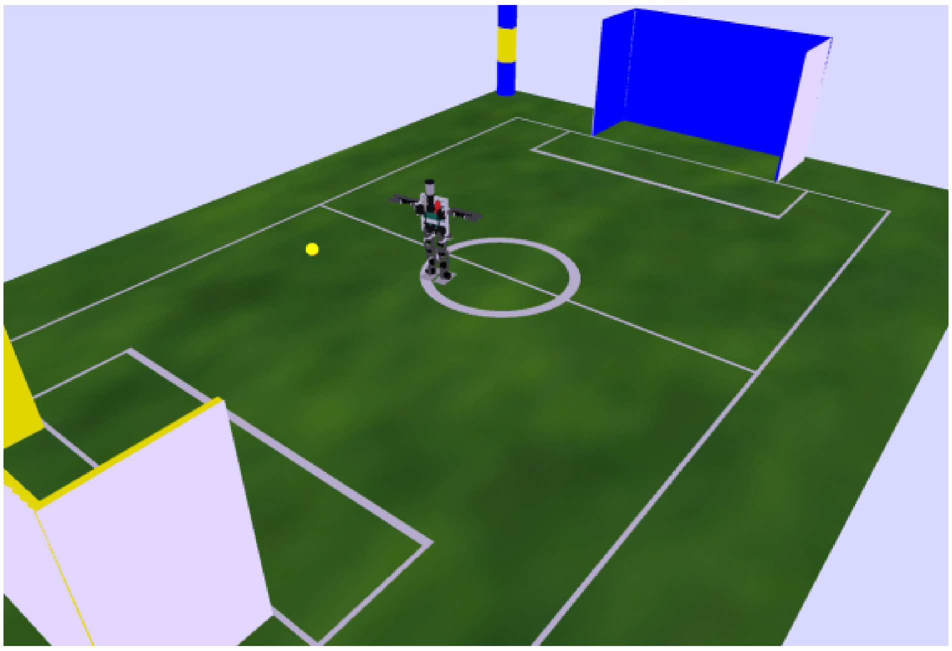


Figure 4.2: Simulation scene

4.2) and the service controlling it.
The scene contains cameras, lights and of course the simulated entities like the field, the robots and the ball.
The service is particularly responsible to set up the simulation and physics engines, to load the entities and generally to manage the simulated scene. As the name says, it is only a environment for the simulated robots. The real robot-related work like walking, sensors, robotcode connection etc. is made in the robot entity described below.

4.1.6 Robot Entity

Just as in the simulation environment, also in the robot entity two parts can be differentiated: a visual part and a service part. The roles are similar: the service has the task to control the visual entity. This section is the focal point of the simulator development.

Visual Entity: physical and graphical model of the robot The model of a robot is strictly spoken a hierarchical union of the particular parts: brackets and servos which are connected by joints (the parts of the simulated robot are shown in figure 4.3). This visual entity is then added to the scene in the simulation environment.

- **Parts**
  A part consists primary of common attributes like position, orientation, weight, friction and others. In addition to this, it has also a physical shape for the world interaction and an optional graphical representation for rendering. The last one

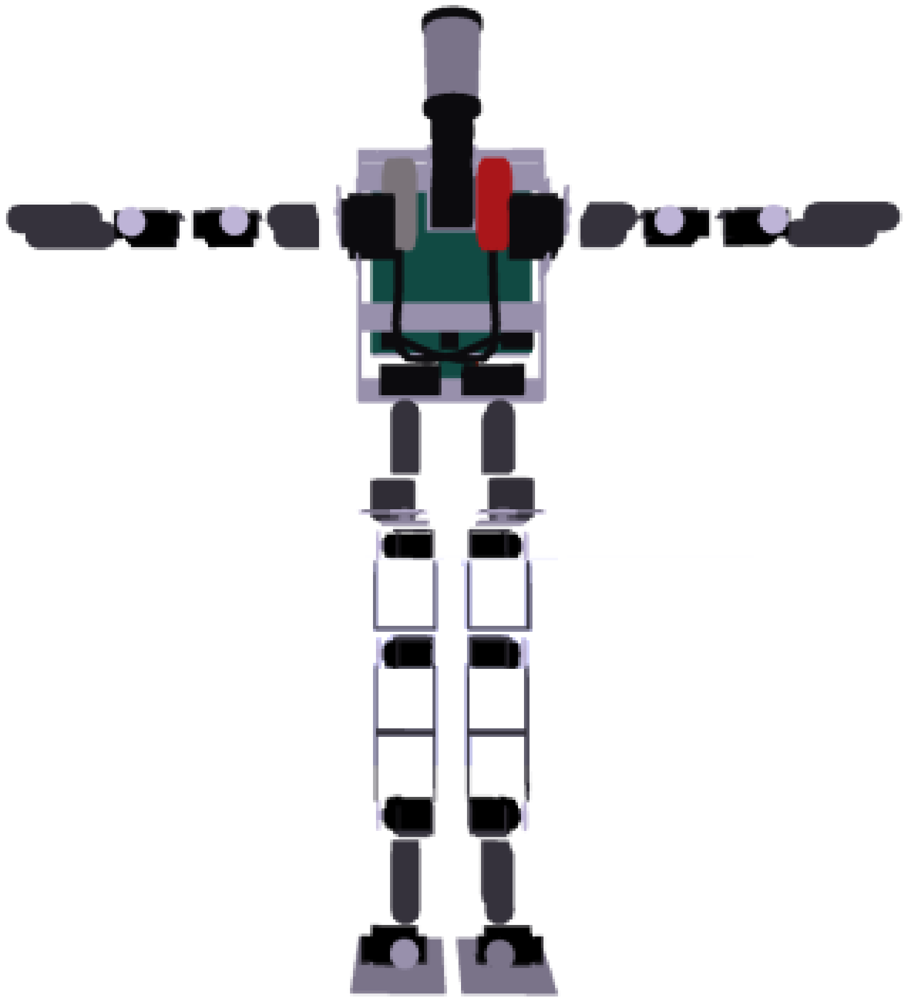


Figure 4.3: Robot model in single parts

can be built from one or multiple primitives: boxes, spheres or capsules. A polygon mesh (3D model) can be optionally assigned to the part for a better look. When there is no mesh defined, the physical primitives are taken for the rendering. Additionally, even shaders can be assigned to the mesh.
For comparison, the figure 4.4 shows the different representations.

A model was used, where the physical shape of a C-bracket consists of three boxes, a servo of only one box.
For the graphical model, first the off-the-shelf models provided by *lynxmotion.com* (supplier of the real-robot brackets) were used. Unfortunately, the models are designed for CAD-Applications and have massively too many polygons, so that the simulation ran with about 10 fps (2.8GHz P4 HT, 1GB RAM, ATI Radeon

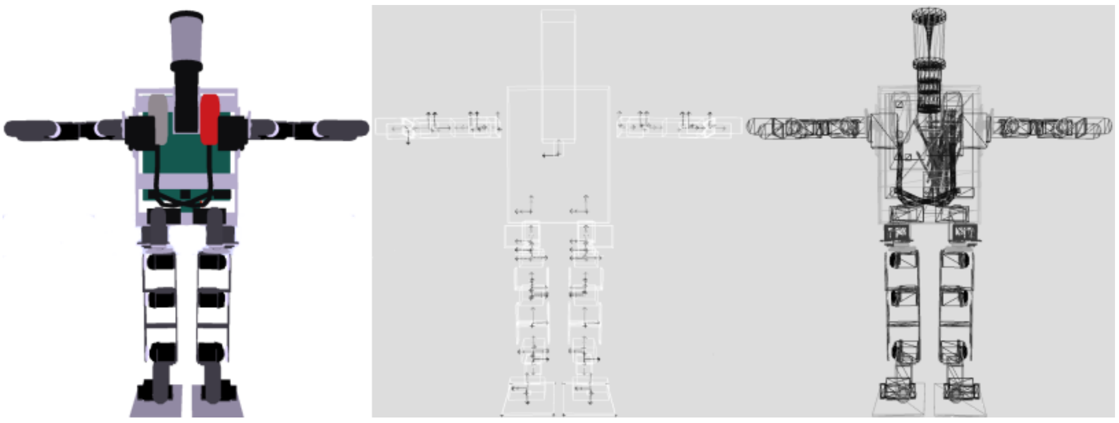


Figure 4.4: Graphical, physical and wireframe representation

X1600). As a consequence they have been immediately replaced by new self-made ones, which have only 5% of the original polygons resulting in framerate of the original 60 fps.

- **Joints**
  Joints are connectors, that specify how two parts can move in respect to each other around that joint. They can connect two parts absolutely stiff and rigid. It can also act as a hinge between them or even like a spring. The most popular joint type in the *AgeiaPhysX* is the 6-degrees-of-freedom (DOF) joint (3 rotational + 3 translational), with the ability to
  - lock/unlock particular DOF
  - determine the spring properties (stiffness, damping)
  - determine the physical correctness (for the sake of computation time)

  Thereby almost every imaginable connection method can be created by combining these properties.

  Here only one rotational degree of freedom is needed, so the others have been locked. The spring properties are set to as-rigid-as-possible, however, at the beginning there were still problems with the wobbliness, so the physical correctness had to be additionally adapted. These properties are set when a joint is created. Nevertheless, the more important part is the connection process of two parts. Two crucial points can be emphasized:
  - **Positioning in world (absolute) coordinates**
    This is the first essential part: placing the brackets and servos correctly in absolute coordinates in the world (with proper orientation). Within these coordinates, these parts are created and inserted into the scene. It is important to already preconceive in this step, how the parts will be connected relatively later on. The reason for this is explained under "Relative positioning".

  - **Connection**
    The next step is the relative connection by the joint. It has two connectors and one part is assigned to each of them. This is the second crucial part: relative positions, rotation axes and the normal vectors (orientation) have to be specified - in respect to certain constraints:
    * **Relative positioning**
      This step determines the real position of a part in the world, since every part is positioned relatively to its parental part. The only part, whose absolute position does matter is the root of that hierarchy, which is the torso in this case. But this also means that the initial *absolute* positioning of the parts mentioned above will be "overwritten" by the relative one when the physical engine will start running. So, if for example the orientation is initially set wrong, the part will be forced to turn with respect to the relative connection. This can cause odd effects like shaking or even falling of the robot.
    * **Rotation axis matching**
      For every part the rotation axis has to be determined. Transformed into world coordinates, the parts both have to lay on the same axis, ensuring they will also turn around the same axis. If this is not done correctly the simulated robot simply explodes.
    * **Normal vector matching**
      Determines the orientation of the part.

  The assembly is finished at that moment in the hard code, making changes in the model not really comfortable. As the real robot changed permanently, it was often needed to also adapt the simulated model, which took quite a lot of time.

- **Other issues when creating the model**
  The bare hierarchical union of parts could result in a moving robot, but for getting the best possible simulation, attention has to be paid to some more constraints:
  - **Proportionality**
    A very important issue is the proportionality of the parts, as the same angles have to work on the real as well as the simulated robot. This criteria has been met pretty successfully, so even the inverse kinematics can be debugged in the simulator.
  - **Weight distribution**
    The next crucial issue is the weight of the parts and the weight distribution across the robot. In spite of the fact that all robot parts were weighed and the weights of the simulated parts were set correctly, the first version of the robot had a greater forward inclination than the real one. Therefore the weight distribution had to be adjusted by displacing the centre of mass a bit backwards.

- **Friction**
  Another, not less inferior concern was to find suitable values for the friction, for both the underground and the feet. Many tests were needed to find the proper values. It is likely that if the robot design will change significantly, the friction parameters will have to be adapted again.

4.1.7 Robot Service

This is the second part of the simulated robot: the model-controlling service layer. The robot service is mainly responsible for managing the connection to the robot-code, the robot movement and the sensors.
A service in Microsoft Robotics Studio can be thought of as special type of module. It contains:

- **Unique identifier**
  Making the service unique in the whole simulation runtime.
- **State**
  Data that describes the current state of the service.
- **Main Port**
  Defines an interface to the service.
- **Handler Methods**
  Processes the messages received on the main port.

For an easier understanding how these things work together, figure 4.5 shows a simple scenario. Service A has a main port at which it supports the messages X and Y (Because only for these messages it contains the appropriate handlers). When a service B sends a message X to the main port of A, the message is forwarded to the handler of X, which processes it and does the proper actions (e.g. changing the state).

In case of the used robot service, the state's data consists of joint angles and connection information on its main port. It supports the *Connection* and *Motion Request* messages. Both are sent to the robot service by the simulation environment service.
As mentioned in the beginning of this section, the robot service is responsible for connection management, robot movement and the sensors. These aspects will be explained more detailed in the following paragraphs.

- **Connection management and networking**
  To establish a connection between a simulated robot and the robotcode, the simulation environment sends a *Connection Request* message with the IP address and port number as parameters to the robot service. This urges the connection request handler to try to establish a TCP/IP connection to the robotcode.
  The main purpose of the communication with the robotcode is sending the proper joint-angle values.

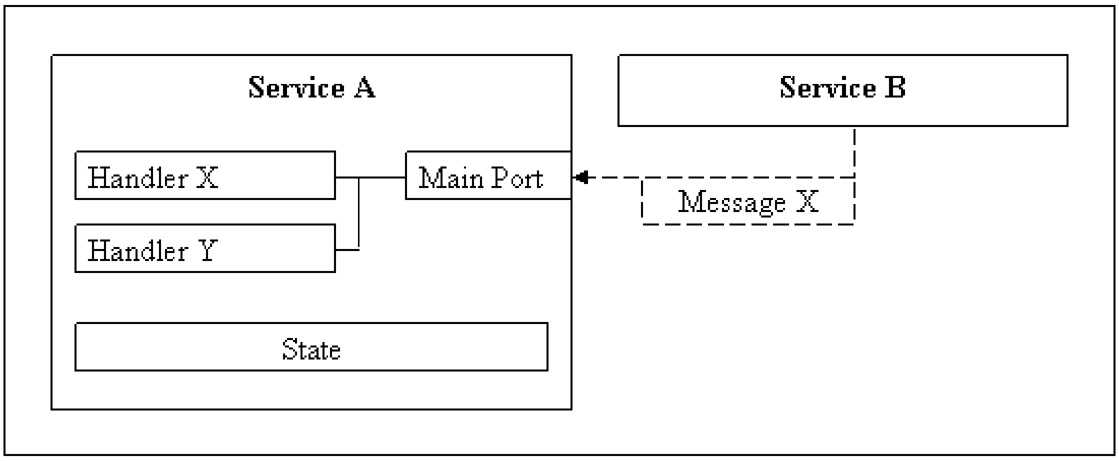


Figure 4.5: Simplified service architecture

So the top-level protocol is a simple XML file formatted as following:

```
<JointAngles>
<J0>-0.000000</J0>
<J1>1.270796</J1>
(...)
</JointAngles>
```

The angles values are sent by the robotcode at about 50Hz, which is absolutely sufficient.

- **Motion**
  The robot movement is also initiated by the simulation environment. It sends a *Motion Request* (a parameterless message) 50 times per second telling the robot service, that it has to read out the joint angles from the robotcode-connection buffer and update the joint positions. 50Hz are absolutely adequate for a smooth movement.
  Moving the joints in *MSRS* is very easy: the right joint has to be picked from the physical engine and the right angle has to be set:

  ```
  ((PhysicsJoint)joints[i]).SetAngularDriveOrientation(targetAngle);
  ```

- **Walking**
  The first version of the Robot Service was supposed to only test the joint movements, so it did not receive motion requests from the robotcode, but simply

played preanimated moves. The sourcecode for this functionality is based on the animation-control code from Laurent Lessieux (originally developed for a simulated Kondo). After adapting it, the robot was able to do push-ups and some cheer-moves. But since the aim was to have a walking robot, the robot service was rewritten for gaining the ability to connect to the robotcode and receive the joint angles from the walking engine. The first tests were really surprising as the simulated robot walked almost stable with the same parameter set as the real robot. Only the step timing had to be adapted, so the movements ran slightly slower and the robot walked. Of course, there were also many problems - mainly through the wobbliness of the joints: under force impact they simply broke the axis. So it were effectively 4-DOF joints with one rotational axis and three translational axes which was quite bad, but on the other hand it somehow simulated the bending process of real brackets - even though a bit too much. Fortunately, in the *MSRS* Forums one of the developers presented a solution for this problem and the translational problem could be fixed with custom degree of correctness. As a result of the improvement, the walk became more stable and the joint wobbliness was pretty realistic if only a little bit of the translational freedom was left.
With the current walking engine the simulated robot can walk omnidirectional, although the rotation causes some troubles - like on the real robot. So the core of the problem is on the one side probably the walking engine itself, but on the other also the friction is likely to cause some problems.

- **Sensors** The next step after having a walking robot was naturally to have a full-featured simulated robot including especially simulated sensors.
  - **Camera**
    The main sensor of the real robot. The hitch in simulating the camera is the fact, that an omnivision camera is used: it features a horizontal 360° view with a vertical opening angle of 120°. But since *MSRS* cannot capture such images, a workaround was needed to be able to get omnidirectional images. In the old simulator such a workaround already existed by computing the omnidirectional image from four normal cameras. Three cameras with 120° FOV are aligned horizontally (three yellow planes on figure 4.6) and one camera is pointing downwards.

    Unfortunately acquiring the images from the simulated cameras did not work properly. At that time no usable documentation existed to troubleshoot this problem, so it was postponed. These troubles were although really problematic, since everything in the framework is based on the camera: self- and ball-localization, as well as the obstacle localization, which are then again of vital importance for behaviour planning.
  - **Oracle**
    For the behaviour planning at least two things are needed: the position of the robot and the ball position. Since they cannot be extracted from the image processor by now, a workaround has been made by "oracling" them. This

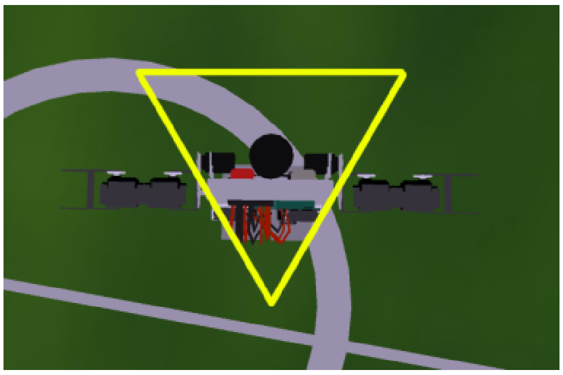


Figure 4.6: Simulatied omnivision camera

means taking their absolute positions directly from the simulator and sending them to the robotcode. This is indeed a good alternative for first rough sketches of the behaviour. A further improvement would be adding noise, which could deliver quite realistic results for more sophisticated behaviour testing.

– **Gyroscope, Acceleration, Inclination**
  Though these sensors are needed for balancing and fall detection in the real robot, in the simulator they would be needed mainly for the fall detection. But they have been not followed up in the current simulator.

4.1.8 Multiple robot simulation

Having a simulated robot being able to walk and to localize itself, a multiple-robot simulation was going to be tested. The main problem is that every simulated robot needs one instance of the robotcode - and one instance of the robotcode and simulator running together on one machine (at least on a 2.8GHz P4 HT with ATI Readon X1600) is an absolute maximum. But due to the simulator and framework design it has not really been a problem. As the connection to the robotcode is established via TCP/IP, the code can also run on other machines across the network. So, on the last days before *Robocup 2007* a simulation with three robots was tested and it ran fine at about 15-20 fps, which is an absolutely acceptable result.

4.1.9 Summary

The first months of using *MSRS* were more experimentation and exploration of the limits of the *Microsoft Robotics Studio* than real development work. Although, the simulator could be already used in some situations: inverse kinematics, walking engine debugging and behaviour planning. After some improvements it will surely become a great simulation environment for a humaniod robot.

4.2 Framework

The motivation for developing a new software framework for the biped soccer robot were the experiences made with the old one during the *German Open 2007*. The old framework is based on the German Team's *AIBO* platform and was ported to *Windows XP* with all bugs and relics. The biggest disadvantage of the old framework was the inability to run concurrency on a multicore platform. So the implementation of a new software, which supports multi-core CPUs and which is designed for *Windows XP Embedded*, was started.

4.2.1 Design

Because the new framework should be able to use multicore CPUs, a threading technology, which is supported by *Windows XP* and which can flexibly assign certain tasks to a specified core, was needed. At the first attempt the *OpenMP Framework* for creating and managing the different threads was tested. The threading class design allows a simple change of the underlying threading technology. Later in the course of development, the threading was switched to *Windows Threads* because it is more flexible than *OpenMP* in stopping, suspending and awaking threads. Furthermore the multi threading capabilities of the new framework should be really modular. Therefore it was split into three parts according to the parts' functionality. The parts are the *Main Program* (see chapter 4.2.2), the *Shared Library* (see chapter 4.2.3) and the *ThreadObjects* (see chapter 4.2.4).

4.2.2 Main Program

The main program implements and manages the threads, called *Processes*, for certain tasks. The main class to mention is the *ThreadManager* which task is to create, configure, startup and destroy threads. The *ThreadBase* is the base class for all *Processes*. It is designed to handle the inter-process communication (IPC) (see chapter 4.2.3) and to manage the *ThreadObjects* which are encapsulated and dynamically loaded with the help of the *ObjectList*.

4.2.3 Shared Library

This library is a static library that contains everything that has to be shared between the different parts of the framework.

Threading

All the threading functionality is combined in the *Thread*-class. Every thread has to inherit from this class and is dedicated to a process type, so they can differ from each other. All threads are event based i.e. they are only executed when a certain event occurs otherwise they are sleeping. These events can be anything. At the moment only timer-based and message-based triggering of the *Processes* is used. The *MotionProcess* for example is triggered by a timer-event every 20ms. Beside time triggering, also message-based triggering is used which means that some *Processes* are started only when a specified message has been received. Message-based triggering guarantees that those threads are only executed when needed.

Inter-Process Communication

Every thread has a receiver list specifying which processes will receive the sent messages. Every receiver needs an incoming queue for received messages. The first implementation used a normal queue, but since the queue had to be a multiple-writer single-reader queue, a lot of work for synchronizing different read and write events was necessary. This synchronization caused a lot of blocking and slowed down the whole framework. The slow-down expressed itself mainly in a stuttering motion since the timing could not be guaranteed. That's why a complete non-blocking system was demanded. To achieve this, the framework now uses ring buffers instead of normal queues. The IPC is realized by a messaging system which is implemented via several non-blocking single reader/writer ring buffers. Every *Process* has one ring buffer for each other *Process* that sends it messages (see chapter 4.2.3). When a *Process* is triggered it collects all readable data from the ring buffer and puts them into its internal legacy queue.

ThreadMessages

ThreadMessage-objects are the only objects which can be used for IPC and network communication. Any data structure or class that is generated in one thread and is used by another one inherits from the abstract class *ThreadMessage*. This class deals with features like thread safety and reference counting (see figure 4.7).

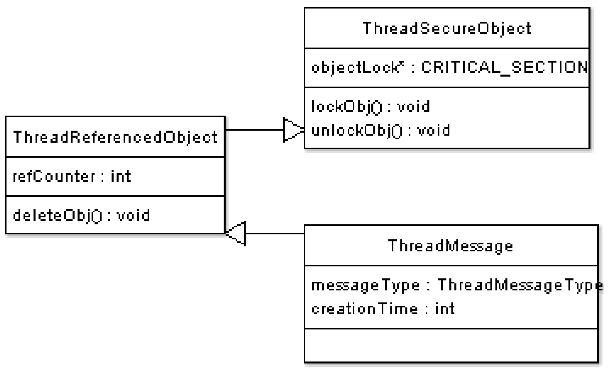


Figure 4.7: *ThreadMessage*

The *ThreadMessages* are dynamically generated and destroyed objects. Dynamically destroyed means that they have an internal reference counter and as soon as this reference counter reaches zero the message is destroyed automatically. This really simplifies the memory management since all a program has to do is counting up and down the references. It must not be aware of the time to destroy the object. Furthermore, all *ThreadMessages* can be serialized (see chapter 4.2.3), which makes it comfortable to send them over the network or save them on harddisk.

Serialization

In a huge software framework it is useful to have the ability to save configuration files for several settings and parameters making recompilation of the whole framework unnecessary. To implement this functionality a flexible serialization scheme was designed. It should be independent from how the data is saved to memory or harddisk, so it is split into two parts, the *Serializable*- and the *Serializer*-class. Both classes are abstract. The *Serializable*-class defines the data of a certain object that should be serialized and the *Serializer*-class defines how this is actually done. For now, there are three serializers implemented. One for saving into a *XML*-File Format, which is primary used for configuration files. A second one saves into a network stream format. The last one is a text serializer that dumps everything into a human readable format for onscreen debugging or logging.

Singletons

The framework contains some data structures that should be accessible from everywhere in the framework. This can be done by using either static classes or singletons. Singletons were chosen because they can be initialized with different values and are only created if they are needed. When dealing with singletons in multithreading environments, it must be guaranteed that only one instance of the specific singleton class will be created. Therefore the code in listing 4.1 is used.

Listing 4.1: Singleton with Double-Check-Idiom

```

Singleton* Singleton::theInstance = 0;
volatile LONG Singleton::instanceLock = 0; //means no instance yet
Singleton& Singleton::getInstance()
{
  while (theInstance == 0) //check 1, non threadsafe
  {
    if (InterlockedCompareExchangeAcquire(&instanceLock,1,0) == 0) //
        check 2, threadsafe
    {
      theInstance = new Singleton();
    };
  }
    return *theInstance;
}
```

Process Types

For all tasks running concurrently or partially parallel, a special process type exists. The following types are the ones needed for the robot. The communication between this threads is shown in figure 4.8.

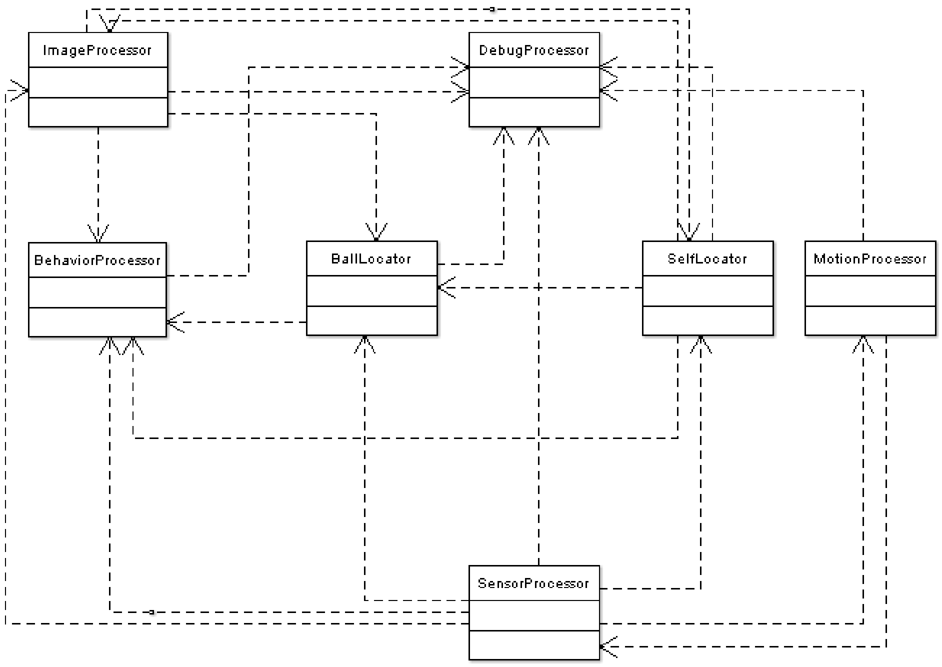


Figure 4.8: Communication between the threads

- **Image Processor**
  One of the main software parts is the image processing. In this thread the images from the two cameras are processed and all percepts will be generated from these images. Most of the calculation time is used by this process.
- **Motion Processor**
  The motion is a time critical thread, because it controls the servos of the robot. To get smooth motions an 50Hz update frequency is needed. This was one of the hardest challenges in designing the framework. *Windows XP Embedded* is not a real time operating system, so no guarantee for a 50Hz scheduling of this thread is given. Every time this thread sleeps, it might take longer than 20ms for awaking it again. One reason for this problem is the synchronization between several threads and another reason are blocking calls to the I/O interfaces by the image and the sensor processor. The best actual solution for now is dealing with a failure rate up to 4 percent i.e. in one second the time limit of 20ms is missed twice at most. To

achieve this performance, ring buffers were implemented for the message system and are given a higher priority.
It is very important to use no debug commands in one of the motion modules, because they effect a blocking situation.

- **Sensor Processor**
  A reason to put the sensor readings and processings in their own thread is the blocking call for reading the sensor values.
- **Behavior Processor**
  The behavior engine does not need an update frequency as high as the motion and it can run concurrently with the image processor.
- **Selflocator and Balllocator**
  These are the processes creating the world models based on the percepts the image processor has generated. While the Selflocator and the Balllocator update their models, the image processing can run in parallel.
- **Debug Processor**
  The debug thread is responsible for the network communication to *RobotControl XP*.

4.2.4 ThreadObjects

The framework provides several threads called *process* which run concurrently and communicate with each other. A modular concept was created for implementing software solutions. Each module is a separate DLL containing a class inheriting from the *ThreadObject*-class. It is assigned to a process type and is given one of the following types. Possible types are *module*, *sensor* and *actuator*. Many modules can be assigned to each *process*. Furthermore a sequence number can be assigned to each *ThreadObject* to determine the execution order which is important to solve dependencies between the modules.

Sensors

Sensors are the modules, which are called first by the thread in each cycle. All modules which have an input from the environment via an hardware device are *sensors* for example cameras or the acceleration sensor. These modules create *ThreadMessages* containing the values read from the sensors and put it into the message queue.

Module

The *modules* are the processing units of the different threads. For example, the *WalkingEngine* is a *module*. The first thing that is done by a *module* is filtering out all *ThreadMessage*-objects in the queue which are not used in the *module*. From needed *ThreadMessage*-objects a backup is kept by the *module*, because it is possible that a certain *ThreadMessage*-object is not received in every cycle. In this case the *module* operates on the backup messages instead.

Actuators

All modules that control an hardware output device are *actuators*. For example, the *servo controller* is an *actuator*. It is responsible for building the servo command out of the target angles delivered from the *motion* and for sending it to the hardware device.

4.2.5 Calibration

The servos are controlled by pulse length signals. Calibrating the servos means to associate pulse lengths and joint angles properly. For the calibration it is assumed, that the relation between the pulse length and the joint angle is linear. The first approach was to store the pulse length of two opposite angles (±90°) and to interpolate linearly between them. To set the proper zero position after the servo is attached to the robot, an offset angle could be defined. The signs of the angles could be set with an additional parameter. The problem with this kind of calibration is, that after the servos are built into the robot, most of them are not able to reach the ±90° positions at all. This means, it is necessary to somehow guess the maximal pulse lengths. Additionally, whenever one of the parameters is changed, the zero position offset has to be recalibrated as well. To make the calibration more reliable and to make it possible to calibrate the assembled robot a separate $referenceAngle$ was defined for every joint. This reference angle can be chosen so that it is easy to reach and measure it for every joint. For each reference angle, the $referenceAnglePulseLength$ is stored. The zero position is no longer defined using an angle offset, but by defining the *zeroPositionPulseLength* directly. The $referenceAnglePulseLength$ and the *zeroPositionPulseLength* can be determined independently. An algebraic sign parameter is no longer necessary, because it is implicitly specified in the sign of the reference angle. The pulse length of a given angle α is calculated as follows

$$pulseLength(\alpha) = \frac{referenceAnglePL - zeroPositionPL}{referenceAngle} \cdot \alpha + zeroPositionPL$$

To make it easier and more accurate to determine the calibration pulse lengths, a calibration device was designed (see figure 4.9). It is possible to attach the robot firm to this device, so that markers on the sides of the device can be used to bring the joints in the appropriate zero or reference angle positions. With the help of the new calibration method and the calibration device, it is now possible to calibrate the robot in less than one hour.

4.3 The Walking Engine

The walking engine used by the DohBots has been developed by Ralf Kosse in 2006 as a part of his diploma thesis. For a detailed description see [Kos06]. This walking engine

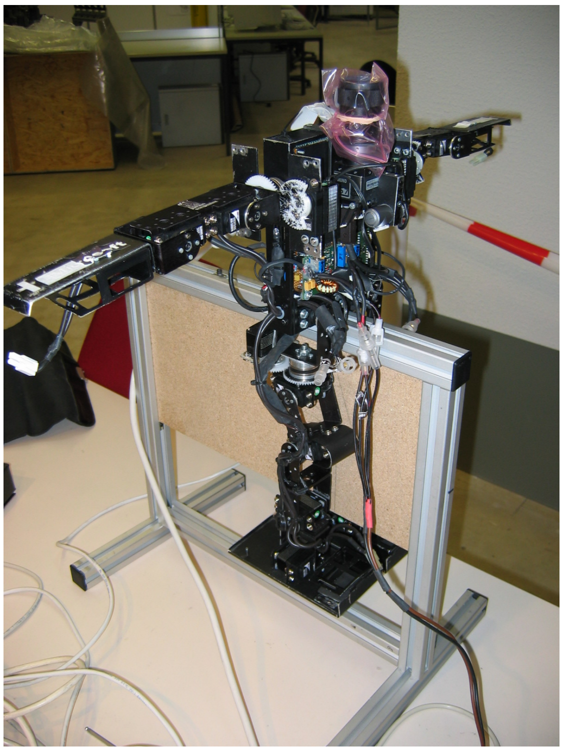


Figure 4.9: The calibration device for the robots

calculates in every frame the value $t \in [0;1]$,

$$t := \frac{currentTime()\%StepDuration}{StepDuration}$$

where *StepDuration* is a parameter. Then it sets the joint angles of each leg according to a predefined trajectory function.

$$JointAngles = trajectory(t)$$

The function *trajectory*() is of a kind, that let the foot move straight backward if $t \in [0;0.5]$ and forward on an adjustable curve if $t \in [0.5;0]$. This works vice versa for the other foot. To calculate the actual joint angles from the trajectories, the function *trajectory*() uses a concept of inverted kinematics. For details on the inverted kinematics see section 4.5. For the arm movements the shoulder joints are set using the sinus function (see figure 4.10). After this calculation, offsets are added to some hip and foot joints to adjust the movement of the body. Beside *StepDuration*, the additional parameters are:

- **StepHeight**
  The distance between the highest point of the foot trajectory and the lowest.

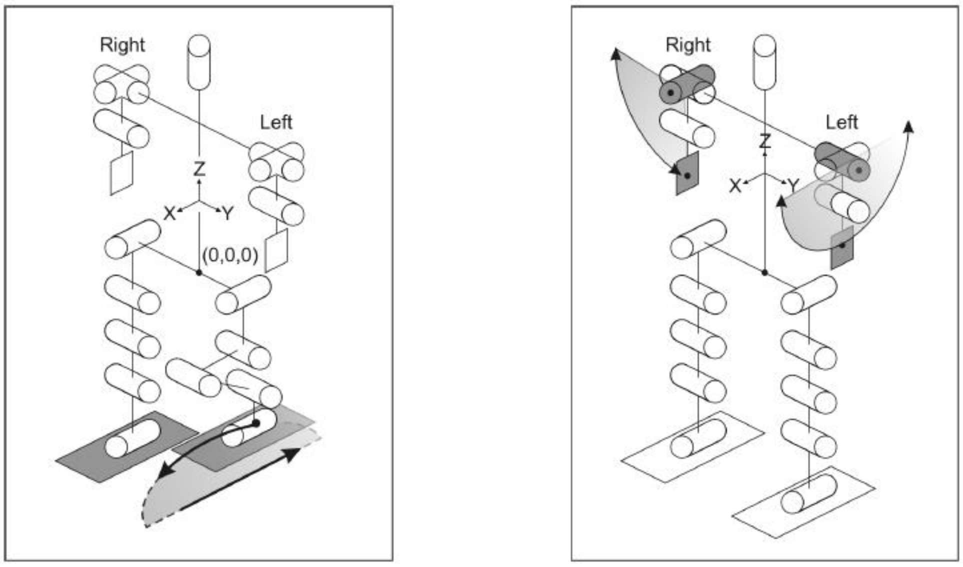


Figure 4.10: Feet and arms of the robot moving along trajectories

- **StepLength**
  The length of each step.
- **ArmAmplitudeX**
  The amplitude of the arm movement in x direction.
- **ArmPhaseX**
  The phase of the arm movement in x direction relative to the movements of the feet.
- **ArmAmplitudeY**
  The amplitude of the arm movement in y direction.
- **ArmPhaseY**
  The phase of the arm movement in y direction relative to the movements of the feet.
- **BodyAmplitudeX**
  The amplitude of the body movement in x direction.
- **BodyPhaseX**
  The phase of the body movement in x direction relative to the movements of the feet.
- **BodyAmplitudeY**
  The amplitude of the body movement in y direction.

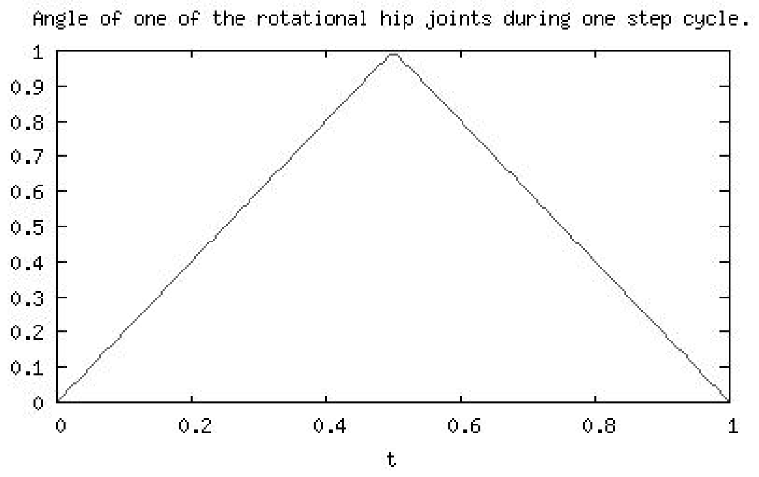


Figure 4.11: Angle of one of the rotational hip joints during one step cycle

- **BodyPhaseY**
  The phase of the body movement in y direction relative to the movements of the feet.
- **YOffset**
  The distance between the feet.
- **ZOffset**
  The whole foot trajectory is moved closer to the robots body by this value.
- **footTiltX**
  Constant offset added to the foot joints to tilt the robot in x direction.
- **footTiltY**
  Constant offset added to the foot joints to tilt the feet inward or outward.
- **AccelerationPerStep**
  Instead of setting the StepLength at once, the StepLength is increased by this value each frame, to make the movement of the robot smoother during acceleration.

As this walking engine was initially designed for a robot that had no joint to rotate the leg around the z axis, it does not provide a move that rotates the robot. To approximate the movement of a robot that turns around, the rotational hip joint of the robot is moved in a position during the time the foot is on the ground ($t \in [0; 0.5]$) and moved back during the time, the foot is in the air ($t \in [0.5 : 1]$) and vice versa for the other foot (see figure 4.11). The advantage of this kind of parametrized walking engine is, that the walk movement can be influenced easily by adjusting the parameters. To be more flexible, the following additional parameters were added during the development of the walking engine:

- **StepLengthLeft and StepLengthRight**
  Separate StepLength for the left and the right foot.
- **ZOffsetLeft and ZOffsetRight**
  Separate Z-Offsets for the left and the right foot.
- **BodyYOffset**
  Offset to constantly move the body a bit to the left or the right.
- **FootScaleFactor**
  A factor multiplied to the movement the foot joints. This makes the foot a bit more (or less) tilt during the time the foot is lifted.
- **bodyXOffset**
  Offset to tilt the body forward or backward.
- **footSpread**
  Offset to spread the feet in a V-like manner.

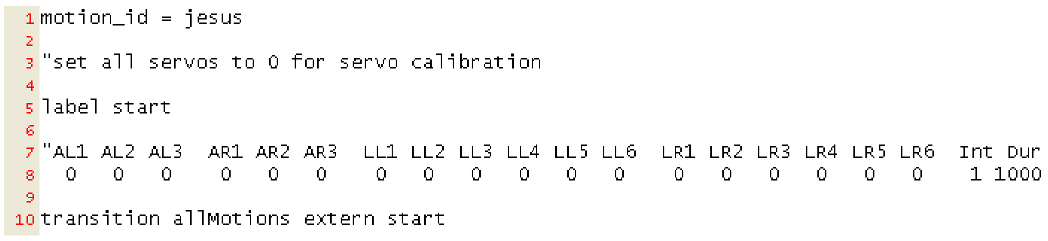
4.3.1 Problems of the walking engine

A number of problems with this walking engine were experienced. First, it is very time consuming to adjust the walking movement if it is wanted to change something on the walk where there is no parameter to do that, because the only way to do this is to add a new parameter directly to the walking engine. Second, this walking engine does not use any sensor information to stabilize the walk. To solve this, a balancing engine was developed (see section 4.6). Third, this walking engine does not know anything about other moves which are performed, for example by the special actions engine. This is a problem as the transitions from and to other movement types (for example kicks) are not controlled and the robot loses stability often.

4.4 Special Actions, Kick, Standup

The Special action engine is responsible for implementing different movements (special actions). Walking is excluded, because it is controlled by the walking engine.
The special action engine had experienced several evolution stages. The old engine consisted of a set of special action files (mof), which had to be described in the source code and a build-in compiler, which created one file (specialaction.dat) out of these mof files. Each new file (new movement) had to be explicitly described in the source code and in a description file (extern.mof). Each change in the files resulted in the necessity of recompiling, because the compiler was build-in. The MOF files could be edited with an text editor, what was sometimes complicated due to the structure of these files.

Each special action file consists of the following parts:



```
motion_id = jesus

"set all servos to 0 for servo calibration

label start

"AL1 AL2 AL3  AR1 AR2 AR3  LL1 LL2 LL3 LL4 LL5 LL6  LR1 LR2 LR3 LR4 LR5 LR6  Int Dur
  0   0   0    0   0   0    0   0   0   0   0   0    0   0   0   0   0   0    1 1000

transition allMotions extern start
```

Figure 4.12: MOF file examaple: jesus.mof

- **motion id**
  The unique identification of the file (motion) in the source code and in the description file (extern.mof). It is used for calling certain special actions (e.g. kickLeft).

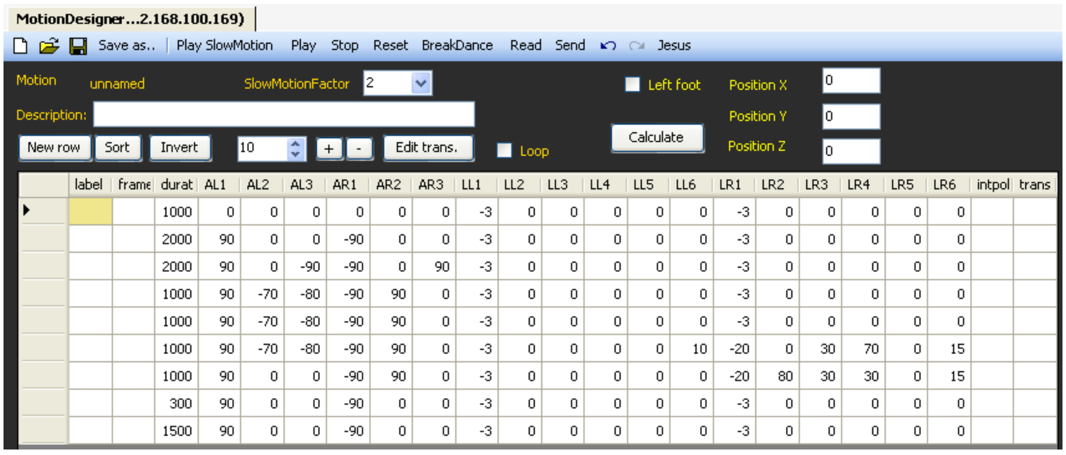
- **Short movement description**
  For example "‘kick the ball with left foot"’. All lines beginning with quotation marks are comments. So the description is optional and can be removed.

- **Movement definition**
  It consists of a set of numbers. The first 18 are joint angles (in degrees). Three angles are for the left arm (AL1 to AL3), followed by three for right arm (AR1 to AR3), then six angles for the left leg (LL1 to LL6) and for the right leg (LR1 to LR6). The last two numbers are interpolation (values between 1 and 3) and duration (in milliseconds) parameters.

To create new movements or edit one, it was necessary to work with each joint separately. Afterwards the whole SpecialActionEngine had to be recompiled. That is the reason why the motion compiler was separated from the SpecialActionEngine and the MotionDesigner was integrated into RobotControlXP (see figure 4.13).
The stand-alone motion compiler allows to recompile only specialaction.dat after a movement was edited. It takes less time than before and can be done on the fly (e.g. if it is necessary to change some parameters for kicking).
The MotionDesigner made working with movements a little bit more comfortable. Additionally it includes some functions from the KinematikEngine, so that it can calculate joint angles out of foot positions. Therefore it has to set the relative foot position (x, y and z). It is also possible to send movements directly to the robot without compiling any special action files.
The last changes made within the SpecialActionEngine was porting the files to XML standard and creating seperate movements for each robot, so that the movements can be optimized individually for the robots.

4.4.1 Kick

The current kick offers 5 different possibilities:



Motion unnamed

SlowMotionFactor 2

Left foot

Position X 0

Description:

Position Y 0

10

Loop

Position Z 0

	label	frame	durat	AL1	AL2	AL3	AR1	AR2	AR3	LL1	LL2	LL3	LL4	LL5	LL6	LR1	LR2	LR3	LR4	LR5	LR6	intpol	trans
▶			1000	0	0	0	0	0	0	-3	0	0	0	0	0	-3	0	0	0	0	0		
			2000	90	0	0	-90	0	0	-3	0	0	0	0	0	-3	0	0	0	0	0		
			2000	90	0	-90	-90	0	90	-3	0	0	0	0	0	-3	0	0	0	0	0		
			1000	90	-70	-80	-90	90	0	-3	0	0	0	0	0	-3	0	0	0	0	0		
			1000	90	-70	-80	-90	90	0	-3	0	0	0	0	0	-3	0	0	0	0	0		
			1000	90	-70	-80	-90	90	0	-3	0	0	0	0	10	-20	0	30	70	0	15		
			1000	90	0	0	-90	90	0	-3	0	0	0	0	0	-20	80	30	30	0	15		
			300	90	0	0	-90	0	0	-3	0	0	0	0	0	-3	0	0	0	0	0		
			1500	90	0	0	-90	0	0	-3	0	0	0	0	0	-3	0	0	0	0	0		

Figure 4.13: MotionDesigner

- **kickLeft**
  A slow kick with the left foot.
- **kickLeftHard**
  A fast and a strong kick with the left foot.
- **kickRight**
  A slow kick with the right foot.
- **kickRightHard**
  A fast and a strong kick with the right foot.
- **kneeKick**
  Very fast and a very strong kick with one of both knees.

kickLeft and kickRight consist of 4 steps:

1. Shift torso to the side
2. Raise kicking foot
3. Kick
4. Move foot back and align

With kickLeftHard and kickRightHard an additional step was inserted:

1. Shift torso to the side
2. Raise kicking foot
3. Bring foot behind

4. Kick
5. Move foot back and align

The weight of the robot and the instability of the joints lead to the fact that it is very difficult to let the robot stand on one foot. Particularly when implementing the kick movement, the robot often fell down. Especially after the robot has walked it was necessary to adapt the kicks. The only solution for this problem was the use of kneeKick. This kick consists of a very fast movement downward (kneel). It hits the ball with the knee and pushes it forward.
In order to have a better and more stable kick in the future, it is important to improve the stability of the robots. In the first place it would be an improvement to lower the robot weight.

4.4.2 Stand up

A stand up movement is very important, because according to rules each robot has to be able to stand up within 45 seconds, otherwise it is penalized. When rising, one can differentiate two movements: standUpFront (if the robot lies on the chest) and standUpBack (if the robot lies on the back).
With standUpBack the robot moves the feet under its hips. So the center of mass is shifted into the center. Afterwards it stands up and tries to keep balance by stabilizing arm movements. With this kind of standup, the knees' servos have a very high load. Thus it is not recommended to repeat the standup movement many times in a short period of time.
With the first version of the robot, it was possible to make standUpFront movement the same way. For the second version it was necessary to change this movement due to the following constructional restrictions. The Degrees of freedom at the knees are forward limited, so that it is not possible for standUpFront to move the feet under the hips. Therefore the standUpFront movement turns the robot on the back. Afterwards the standUpBack movement is triggered.

That has some disadvantages in the comparison with normal stand up movements:

- Standing up from front takes now twice the time as before.
- It leads to a higher power consumtion.
- During turning on the back, the robot falls down and could be damaged.
- During turning on the back, the robot shifts its body to the side. If there is not enough free space, the movement can not be proceeded.

In order to be able to improve these movements in the future, it is necessary to make some changes to the construction of the robot (e.g. less weight, longer arms, better knees).

4.5 Inverted Kinematics

Inverted kinematics means the problem of calculating the angle positions $\alpha_1, ..., \alpha_n$ of a robots arm or leg from a desired foot or hand position $\vec{x}$. Be aware, that $\vec{x}$ can consist beside the space coordinates x, y and z of more degrees of freedom describing the orientation of the hand or foot in space. The walking engine used by the *Doh'Bots* uses an approach to calculate the leg angle values from the foot position directly, using the sinus and cosinus theorem. By nature, this way of calculation is very fast. The disadvantage is, that in this calculation it is assumed, that the foot is parallel to the x, y plane of the robots coordinate system and that the foot is not rotated around the z axis. Because of this, it was necessary to design and implement algorithms, that calculate the angle values of a foot position in an iterative way. Starting from an initial set of joint angles $\vec{\alpha} = \alpha_1, ..., \alpha_n$ the foot position

$$f(\vec{\alpha})$$

is calculated using forward kinematics and from this position a new set of angle values $\vec{\alpha}'$ is calculated, so that its related foot position is closer to the desired position $\vec{x}_0$.

$$||f(\vec{\alpha}') - \vec{x_0}|| < ||f(\vec{\alpha}) - \vec{x_0}||$$

This step is repeated, until the reached foot position is "close enough" to the desired one. In the next section the method to calculate the forward kinematic is described. The next two sections describe the two algorithms to find "better" angle values from a given set of angle values.

4.5.1 Forward kinematics

To calculate the foot position from the joint angles the affine transformation is considered, that transfers vectors from the foot coordinate system Σ' to the robot coordinate system Σ.

$$\varphi : \Sigma' \to \Sigma$$
$$\varphi(\vec{x}') = \vec{x}$$
$$\vec{x} = A(\vec{x}') + \vec{b}$$

Where A is a rotational matrix and $\vec{b}$ a vector. Another way to write it down is:

$$T := \begin{pmatrix} a_{11} & a_{12} & a_{13} & x \\ a_{21} & a_{22} & a_{23} & y \\ a_{31} & a_{32} & a_{33} & z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} A & \vec{b} \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \vec{x} \\ 1 \end{pmatrix} = T \begin{pmatrix} \vec{x}' \\ 1 \end{pmatrix}$$

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_1 & -\sin\alpha_1 & \frac{1}{2} lengthBetweenLegs \\ 0 & \sin\alpha_1 & \cos\alpha_1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\begin{pmatrix} \cos\alpha_6 & -\sin\alpha_6 & 0 & 0 \\ \sin\alpha_6 & \cos\alpha_6 & 0 & 0 \\ 0 & 0 & 1 & -hipLength \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\begin{pmatrix} \cos\alpha_2 & 0 & \sin\alpha_2 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha_2 & 0 & \cos\alpha_2 & -upperLegLength \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\begin{pmatrix} \cos\alpha_3 & 0 & \sin\alpha_3 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha_3 & 0 & \cos\alpha_3 & -lowerLegLength \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\begin{pmatrix} \cos\alpha_4 & 0 & \sin\alpha_4 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha_4 & 0 & \cos\alpha_4 & -upperFootHeigth \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\begin{pmatrix} 1 & 0 & 0 & \frac{1}{2} footLength - footCenterToRear \\ 0 & \cos\alpha_5 & -\sin\alpha_5 & footCenterToOuter - \frac{1}{2} footWidth \\ 0 & \sin\alpha_5 & \cos\alpha_5 & -lowerFootHeight \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 4.14: The transformation, that transfers coordinates from the left foot coordinate system to the robot coordinate system

To write this transformation down, it is enough to write down the matrices $T_{\alpha_1}, ..., T_{\alpha_6}$ of the single joints and calculate:

$$T = T_{\alpha_1} T_{\alpha_6} T_{\alpha_2} T_{\alpha_3} T_{\alpha_4} T_{\alpha_5}$$

There are many ways to parametrise the matrix A and the vector $\vec{b}$. For example rotation matrices, Denavit-Hartenberg, Euler angles, RPY-angles, etc. Because rotation matrices have a very simple form, if they describe a rotation around one of the coordinate system axes and the joints of the robot all rotate around one of them, the decision was made to use simple rotation matrices. For the left leg of the robot the transformation is written down in figure 4.14. Compare with figure 4.16. The transformation for the right leg looks very similar. The x, y and z coordinates of the foot can now be calculated:

$$\vec{x} = \varphi(\vec{0}) = \vec{b}$$

The matrix A gives information about the orientation of the foot. Compare it with the transformation matrix of the RPY-transformation:

$$A_{RPY} = \begin{pmatrix} \cos\beta\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \cos\beta\sin\gamma & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma \\ -\sin\beta & \sin\alpha\cos\beta & \cos\alpha\cos\beta \end{pmatrix}$$

and see, that the rotations around x, y and z axes α, β and γ can be calculated as follows

$$\begin{aligned} \beta &= \arcsin -a_{31} \\ \alpha &= \arcsin\left(\frac{a_{32}}{\cos(\arcsin -a_{31})}\right) \\ \gamma &= \arcsin\left(\frac{a_{21}}{\cos(\arcsin a_{-31})}\right) \end{aligned}$$

In section 4.5.2 the Jacobi matrix of the function:

$$f : U \subseteq \mathcal{R}^6 \mapsto \mathcal{R}^6$$

$$f(\vec{\alpha}) = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix} := \begin{pmatrix} x \\ y \\ z \\ \alpha \\ \beta \\ \gamma \end{pmatrix}$$

is needed. The partial derivatives of f:

$$\begin{aligned} \frac{\partial f_1}{\partial \alpha_i} &= \frac{\partial x}{\partial \alpha_i} \\ \frac{\partial f_2}{\partial \alpha_i} &= \frac{\partial y}{\partial \alpha_i} \\ \frac{\partial f_3}{\partial \alpha_i} &= \frac{\partial z}{\partial \alpha_i} \end{aligned}$$

$$\frac{\partial f_4}{\partial \alpha_i} = \frac{\frac{\partial a_{32}}{\partial \alpha_i}\cos(\arcsin -a_{31}) + a_{32}a_{31}\frac{\partial a_{31}}{\partial \alpha_i}\frac{1}{\sqrt{1-a_{31}^2}}}{\sqrt{1-\left(\frac{a_{32}}{\cos(\arcsin -a_{31})}\right)^2}\cos^2(\arcsin -a_{31})}$$

$$\frac{\partial f_5}{\partial \alpha_i} = -\frac{\partial a_{31}}{\partial \alpha_i}\frac{1}{\sqrt{1-a_{31}^2}}$$

$$\frac{\partial f_6}{\partial \alpha_i} = \frac{\frac{\partial a_{21}}{\partial \alpha_i}\cos(\arcsin -a_{31}) + a_{21}a_{31}\frac{\partial a_{31}}{\partial \alpha_i}\frac{1}{\sqrt{1-a_{31}^2}}}{\sqrt{1-\left(\frac{a_{21}}{\cos(\arcsin -a_{31})}\right)^2}\cos^2(\arcsin -a_{31})}$$

are calculated.

In all cases, the partial derivatives of the members of A and $\vec{b}$ are needed. The matrix $\frac{\partial T}{\partial \alpha_i}$ of the partial derivatives of the members of T can be calculated.

$$\frac{\partial T}{\partial \alpha_i} = T_1 ... T_{i-1} \frac{\partial T_i}{\partial \alpha_i} T_{i+1} ... T_6$$

All these calculations are only possible, if α, β and γ are not equal $\pm 90°$. If these angles are close to $\pm 90°$ the algorithm gets unstable as well. If for example an angle γ of around $\pm 90°$ is needed, it should be calculated like this. [14]

$$\gamma = \arccos\left(\frac{a_{11}}{\cos(\arcsin - a_{31})}\right)$$

$$\frac{\partial f_6}{\partial \alpha_i} = -\frac{\frac{\partial a_{11}}{\partial \alpha_i}\cos(\arcsin - a_{31}) + a_{11}a_{31}\frac{\partial a_{31}}{\partial \alpha_i}\frac{1}{\sqrt{1-a_{31}^2}}}{\sqrt{1-\left(\frac{a_{11}}{\cos(\arcsin - a_{31})}\right)^2}\cos^2(\arcsin - a_{31})}$$

To reach a desired foot position $\vec{x}$ as close as possible, it is necessary to define what a "distance" in the 6-dimensional position space means. Consider the function

$$\begin{gathered} ||f|| : U \subseteq \mathcal{R}^6 \mapsto \mathcal{R} \\ ||f(\vec{\alpha})|| \end{gathered}$$

with $||\cdot||$ being a six dimensional norm. As for the robot, joint angles are measured in radians and the robots dimensions in mm, the decision was made to use the following norm.

$$\left\| \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix} \right\| := \sqrt{[f_1^2 + f_2^2 + f_3^2] + 180 \cdot [f_4^2 + f_5^2 + f_6^2]} \tag{4.1}$$

This norm weights a rotation angle of $0,1$ radians equal to a distance of $18mm$. In chapter 4.5.3 the gradient of this function is needed. To calculate it, let us write down the partial derivatives of $||f||$.

$$\frac{\partial ||f||}{\partial \alpha_i} = \frac{1}{||f||}\left[f_1\frac{\partial f_1}{\partial \alpha_i} + f_2\frac{\partial f_2}{\partial \alpha_i} + f_3\frac{\partial f_3}{\partial \alpha_i} + 180\left(f_4\frac{\partial f_4}{\partial \alpha_i} + f_5\frac{\partial f_5}{\partial \alpha_i} + f_6\frac{\partial f_6}{\partial \alpha_i}\right)\right]$$

14 The standard C library supports a method $atan2(x, y)$, that does such case switches by default. It is possible to calculate the angles the following way:

$$\begin{aligned} \beta &= atan2(-a_{31}, \sqrt{a_{11}^2 + r_{21}^2}) \\ \alpha &= atan2\left(\frac{a_{21}}{\cos(\beta)}, \frac{a_{11}}{\cos(\beta)}\right) \\ \gamma &= atan2\left(\frac{a_{32}}{\cos(\beta)}, \frac{a_{33}}{\cos\beta}\right) \end{aligned}$$

The partial derivatives of the f_i are calculated as above.

4.5.2 Newton algorithm

To calculate a "better" set of angles $\vec{\alpha}'$ of a given set $\vec{\alpha}$, the decision was made to use the so called *Newton* algorithm. First, calculate the function f and the Jacobi matrix J of the function f at the position $\vec{\alpha}$ (see section 4.5.1). Assuming that f is nearly linear close to the desired foot position $\vec{x}_0$ and the current position $f(\vec{\alpha})$ is not too far away from this position, it is possible to calculate a new set of angles by solving the following linear equation system.

$$f(\vec{\alpha}) + J(\vec{\alpha}' - \vec{\alpha}) = \vec{x}_0$$

As f is not really linear, the new position $f(\alpha')$ is not exactly the desired position and it is necessary to repeat this step until the position is "good enough".

$$||f(\alpha') - \vec{x}_0|| < \epsilon$$

For the robot $\epsilon = 0,1$ was chosen. This means a maximal deviation of $0,1mm$ or about $5 \cdot 10^{-4}$ radians. This small tolerance was necessary, because the distances between foot positions of two frames are off this order. This calculation is not possible, if J is singular, what is for example the case, if the knee is completely stretched (the knee angle is zero). Also, this algorithm fails, when the desired position is not reachable at all, or if limitations for one ore more angles need to be exceed to reach the desired position. If this is the case, a so called *quasi-Newton* Algorithm is used instead.

4.5.3 Quasi-Newton algorithm

When minimizing the function $||f||$ (see equation 4.1), it is no longer possible to assume that it is linear, because the gradient vanishes close to a minimum. Beside the gradient now consider the Hesse-matrix at the position $\vec{\alpha}$.

$$||f(\vec{\alpha}')|| \approx ||f(\vec{\alpha})|| + (\vec{\alpha}' - \vec{\alpha})\nabla||f(\vec{\alpha})|| + \frac{1}{2}(\vec{\alpha}' - \vec{\alpha})^T H(\vec{\alpha})(\vec{\alpha}' - \vec{\alpha})$$

To find the minimum of this approximating function, calculate the gradient and set it to zero.

$$\nabla||f(\vec{\alpha}')|| \approx \nabla||f(\vec{\alpha})|| + H(\vec{\alpha})(\vec{\alpha}' - \vec{\alpha}) = 0$$
$$\Rightarrow \vec{\alpha}' = \vec{\alpha} - H^{-1}(\vec{\alpha})\nabla||f(\vec{\alpha})||$$

The chosen way to improve this result was to take $\vec{r}(\vec{\alpha}) := H^{-1}(\vec{\alpha})\nabla||f(\vec{\alpha})||$ as search direction and minimize the now one dimensional function

$$g(t) := ||f(\vec{\alpha} - t \cdot \vec{r}(\vec{\alpha}))||$$

using the bisection method. The problem is to determinate the inverted Hessian matrix of the function $||f||$. To solve this, the decision was made to use an approximated inverted Hessian matrix starting with the unity matrix instead, and try to improve

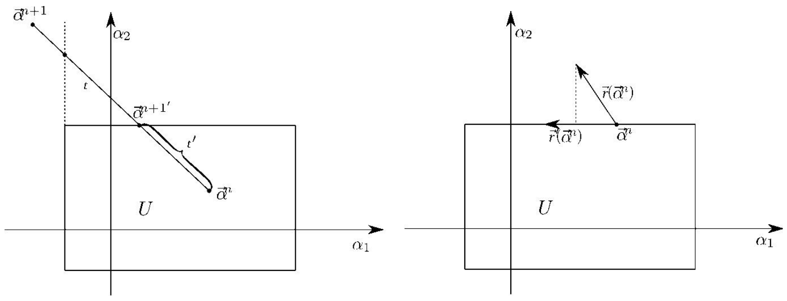


Figure 4.15: Modify the search range and direction in case of exceeding angle limits.

the approximation of the inverted Hesse matrix in every step of the iteration. There are many ways to calculate a approximated Hessian matrix. The *Broyden-Fletcher-Goldfarb-Shanno* method was found to be sufficient. In every step the inverted Hessian matrix is updated:

$$H^{-1'} = \left(E - \frac{\vec{s}\vec{y}^T}{\vec{y}^T\vec{s}}\right) H^{-1} \left(E - \frac{\vec{y}\vec{s}^T}{\vec{y}^T\vec{s}}\right) + \frac{\vec{s}\vec{s}^T}{\vec{y}^T\vec{s}}$$

With E being the unity matrix, $\vec{s} := \vec{\alpha}' - \vec{\alpha}$ and $\vec{y} := \nabla||f(\vec{\alpha}')|| - \nabla||f(\vec{\alpha})||$

For the robot, the joint angles can not reach all values. For every joint angle α_i there are α_i^{min} and α_i^{max} so that

$$\alpha_i^{min} \leq \alpha_i \leq \alpha_i^{max}$$

To make sure, that the joint angles calculated by this algorithm are in these boundaries, there is a check in every step, if the new position

$$\alpha_i' = \alpha_i - t \cdot r_i(\alpha)$$

fits this limitations. If not, reduce t until $\alpha_i' = \alpha_i^{min/max}$.

$$t' = \frac{\alpha_i^{min/max} - \alpha_i}{r_i(\vec{\alpha})}$$

See figure 4.15 left. If a limit is reached in one step, a search direction which does not lead us back into the allowed region in the next step would stuck us at the boarder. In this case it is necessary to change the search direction. By setting the appropriate component of the search direction to zero, the minimum is searched along the border of the search space (see figure 4.15 right).

4.6 BalancingEngine

The *SpecialActionEngine* and the *WalkingEngine* were designed as non-feedback controllers, this means the engines only use the current internal state (consisting of time,

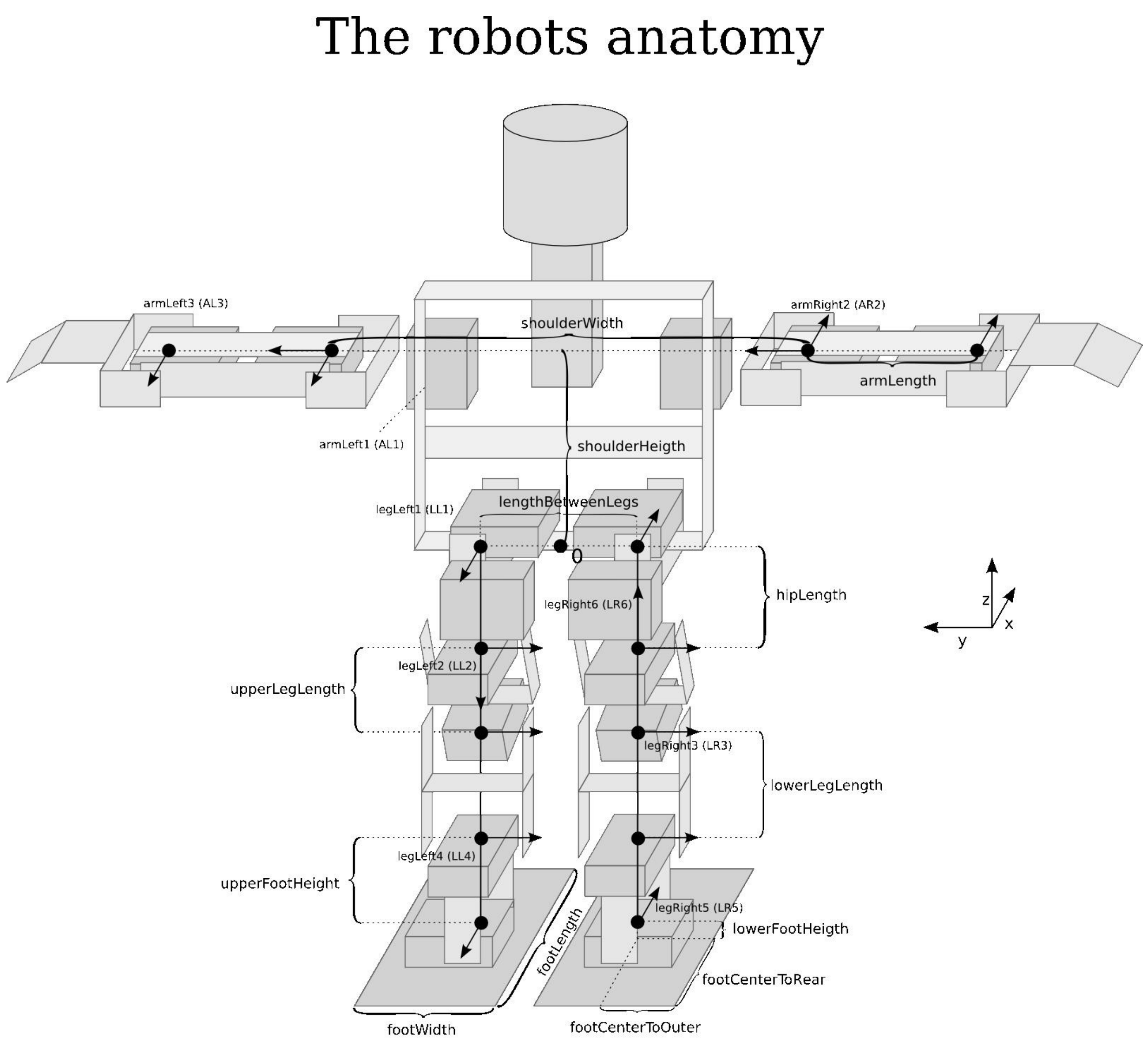


Figure 4.16: The anatomy of the robots.

motion request and other information) and its model of the system to control the system. This works, if the endogenous interferences and exogenous influences (e.g. pushing) are tolerable.

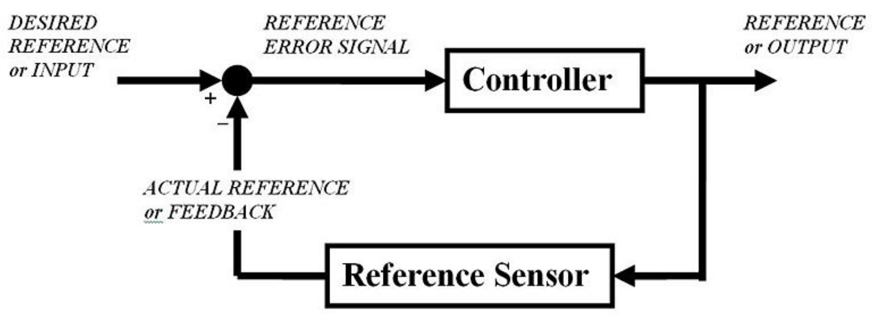


Figure 4.17: Block diagram of a feedback controller (image source Wikipedia)

A feedback controller (see figure 4.17) can be used to increase the stability (or other characteristics) of the system by using sensor information to correct discrepancies between the desired reference and the real reference.
A separate *BalancingEngine* was created to support motion engines that do not use sensor information. The target was to increase stability and robustness of the moving robot.

4.6.1 PID based BalancingEngine

This *BalancingEngine* uses a PID controller (see figure 4.18) to control a reference signal. The supported reference signals are angular rate and inclination, which should be set accordingly (e.g. derived from an internal model) by the currently active motion engine. There are two separate PID controllers for the value around the x-axis (roll) and y-axis (tilt) of the upper body.
The proportional-integral-derivative (PID) controller is a widely used controller. A PID controller uses an error signal $e(t)$ to calculate a correction value $u(t)$ using the given formula

$$u(t) = K_p e(t) + K_i \int e(\tau) d\tau + K_d \frac{de}{dt}$$

where K_p, K_i and K_d are the proportional, integral and derivative gain which are used to configure the controller.
The desired reference used by the controller can be set by the motion engines. If the values are not set then it is assumed that the upper body should be vertical (no inclination) or the angular rate should be zero.
The mapping of the roll and tilt correction values to the joints is done by multiplying the correction value with a joint specific factor and then adding the new correction to the

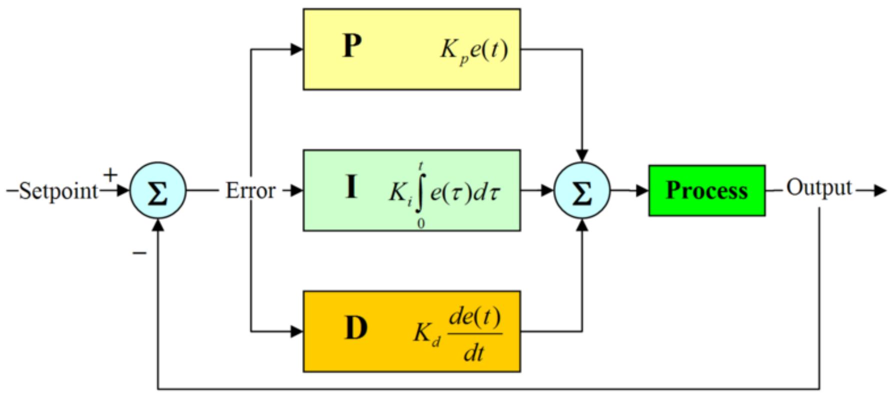


Figure 4.18: Block diagram of the PID controller (image source Wikipedia)

reference joint angles. This realizes the feedback loop. The factors are created manually and are chosen depending on the motion request and the estimated base foot.
The *BalancingEngine* was mainly tested using the *Bosch SMB 380* acceleration sensor to estimate the inclination. The estimated inclination was therefore depending on the measured acceleration, which itself depends on the gravity and the other forces acting on the robot, therefore it was expected to be a good reference signal for the balancing task.
There are, however, a few problems using this input signal. There are distortions due to the elastic link-joint structure, leading to unpredictable aberrations to the reference signal. The integral part was not used, because the error is inconstant, changing fast and heavily dependent on the motion. These problems are amplified if the reference signal is not set by the motion engine and therefore a static reference signal is used.
Another problem is that the servo controllers in the joints permanently try to correct the position of the mechanical system. This results in "shaking" of the robot. The acceleration sensor is very sensitive to this "shaking". This leads to an oscillation in the error signal and can make the robot swing, which gets increased by the corrections of the PID controller. This is especially the case if the other forces acting on the robot are small.
To solve the last problem a set of functions, called "attenuation functions", can be used to decrease the reaction for small correction values and react normal or amplified to higher correction values. The attenuation functions turned out to solve this problem.
An example for an attenuation function is

$$a(x) = \begin{cases} sign(x) * |x|^{exp} * factor, & \text{if } |x|^{exp} * factor \leq max \\ max, & \text{if } |x|^{exp} * factor > max \end{cases}$$

where x is an angle given in radiance, $factor$ is a scaling factor, exp is the main parameter to control the function and chosen $exp > 1$. max defines a switch between the two cases, it also restricts the reaction in this variant. Another variant is to switch to another function (e.g. id), if $|x|^{exp} * factor > max$ is true, then attention should be laid on the scaling factor to create a smooth transition between the two cases.

4.6.2 Conclusion

Overall it is to say that it is a complex task to find a parameter set for the PID based *BalancingEngine*. The usage of the attenuation functions solved one problem, but lead to even more parameters that have to be considered. To find a configuration that works in all situations is problematic, e.g. a solution for walking can lead to swinging while standing. The parameters highly depend on the underlying motion engine and any change there interferes with the parameter set. Even minor changes in a motion engine can lead to the requirement of a new parameter set. Whereas more robust parameter sets have only a very limited positive influence on stability.
For the future it is recommended to replace the direct control of the joint angles and replace it by controlling a suitable value (e.g. *Center of Mass* (*CoM*), velocity of *CoM*) and indirectly controlling the joint angles (e.g. calculate the joint angle configuration to fit the desired *CoM* position). Therefore the creation of an underlying model was started and *BLAS*/*LAPACK* was added to provide a fast linear algebra solution for the upcoming more complex calculations.
There is a need to find good objective measures to evaluate the effect of balancing engines. The subjective visual evaluation turned out problematic and the tuning of parameters could be improved using systematic methods based on objective measures.

4.7 SensorDataProcessor

The used robot platform offers a variety of sensors. A possible sensor configuration utilizes one 3-axis inclination sensor, two 1-axis angular rate sensors, one 3-axis acceleration sensor and four force sensors per foot.
The inclination, angular rate and acceleration sensors are positioned in the upper body of the robot. The *SensorDataProcessor* gets the raw data, which needs to be converted to common units in the corresponding drivers, from all connected sensors. It has the tasks to combine the redundant information, apply basic filtering and basic calculations on the provided data.

4.7.1 Sensor fusion

The information from the inclination, angular rate and acceleration sensors are correlated. In fact the angular rate is the first derivative of the inclination and the acceleration is approximately equal to the first derivative of the angular rate. This can be used to suppress noise from single information sources by combining these information.
The acceleration sensor can be used to directly calculate an approximation for the inclination. It is assumed that gravity, when averaged over a certain time interval, is

the main force acting on the robot, thus the measured acceleration points to the earth center, indicating the inclination of the robots upper body.
All these values can be used to calculate the inclination using an affine combination, the weights of the affine combination can be set in the configuration file.

4.7.2 Falldetection

One of the main tasks of the *SensorDataProcessor* is the falldetection. It has to detect whether the robot is standing or lying and if the robot is lying then on which side. This is important to enable the behaviors to trigger the specific stand-up movement.
Multiple falldection variants were created to support different sensor combinations:

- Inclination sensor only
- Acceleration sensor only
- Acceleration sensor and force sensors

The force sensors are used to detect whether at least one foot of robot is touching the ground or not. The force sensors are the only sensors which are able to provide this information.
The inclination sensor has a special characteristic which had to be considered. The sensor has a fix measurement range and if the real value leaves the supported interval then the minimum i_{min} respectively maximum i_{max} value of this interval is delivered, disturbed by a small amount of noise. If the robot is lying then the measured value v is $v \leq i_{min} + \epsilon$ or $v \geq i_{max} - \epsilon$ over a longer time interval. This characteristic is used to detect that the robot is lying. The measured values are then used to detect on which side it is lying. ϵ can be set in the configuration file with respect to the noise.
The acceleration based falldetection first checks if the measured acceleration on the z-Axis is over a specified threshold (e.g. -0.5g), this can only happen if the inclination of the robots upper body exceeds a value, which depends on the threshold, or if the robot accelerates upwards. If this is the case, then the robot is assumed to have fallen down and the side on which the robot is lying is calculated using the gravity.
The falldetection uses own filters to smooth the measured values over a relatively long time. Working solutions used average filters over a timespan between one and two seconds.

4.7.3 Center of Pressure calculation

The *SensorDataProcessor* supports the calculation of the *Center of Pressure* (*CoP*). For the application the center of pressure and the *Zero Moment Point* (*ZMP*) are equal (see [Gos99]), the term *Center of Pressure* is used here, because it describes exactly the performed calculation.
Due to the nature of the force sensors only the normal forces (see figure 4.19) are measured, this means that no information about tangential forces are available, but the

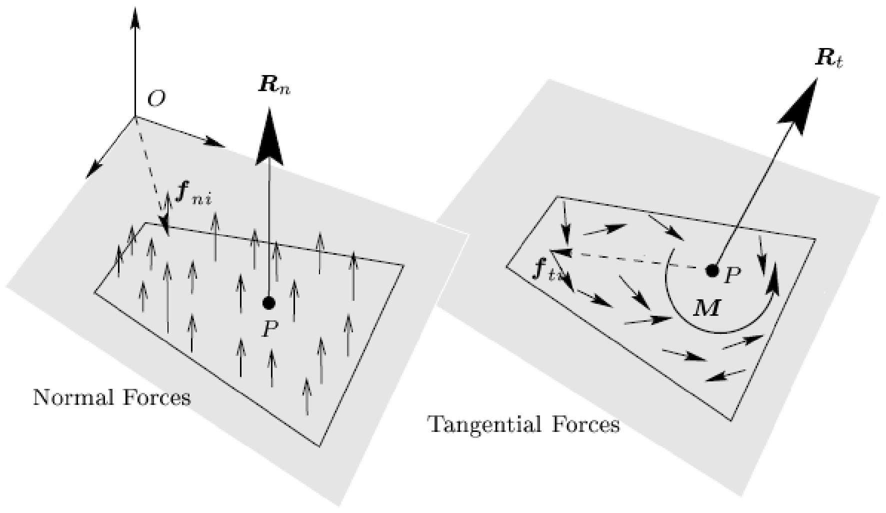


Figure 4.19: Normal and tangential forces affecting a foot (image was modified from [Gos99])

normal forces are sufficient to calculate the position of the *CoP* inside the support polygon. The support polygon is the convex hull of the points of the robot that have contact with the ground, in this case only contact between feet and ground is considered. In this case the *ZMP* and the *CoP* deliver equal information.
Let f_{ni} be the measured normal force at force sensor i. n is the number of force sensors in the feet. $R_n = \sum f_{ni}$ is the (normal) force resultant at the *CoP*. A force resultant can be calculated for each foot by only considering the sensor information delivered from sensors in one foot, this information can be used to detect the base foot. Let p_i be the position of sensor i in the robot coordinate system. The position of the force sensors in the robot coordinate system p_i can be calculated using forward kinematic, if the position of the force sensors under the feet is given.
The position of the *CoP* p_{cop} can then be calculated as:

$$p_{cop} = \frac{1}{R_n} \sum_{i=1}^{n} f_{ni} * p_i$$

4.7.4 Calculation of the camera matrix

Another main task is the calculation of the camera matrix which is later used and refined by the *ImageProcessor*. The camera matrix calculation uses the angles which are read out of the joint servos. The reference target joint angles are used as approximation, if the servo feedback is unavailable.
The camera matrix is calculated using the robot dimensions, information about the link-joint structure and the current joint angles using affine transformations (forward

kinematic). The base foot, which is important for the previous kind of calculation, can be detected comparing the force resultants of both feet. The foot with the higher force resultant is assumed to be the base foot. If the force sensors are unavailable, then the foot with the lowest translation on the z-Axis is assumed to be the base foot.

4.8 Behaviour State Machine

4.8.1 XTC

As the old framework was used at first, the formerly used behaviour concept based on the XTC Language, had to be migrated to the new project. This had the advantage of being used before in the AIBO framework, it just had to be slightly adapted. Additionally, there were lots of experiences with this concept of modelling behaviour state machines.
But with the change to the new framework (see 4.2.2) after the *German Open 2007*, a lot of difficulties with porting and implementing the basis of *XABSL-Engine*[15] and XTC occurred. For this reason, it was decided to discard the XTC engine and to implement a new state machine concept.

4.8.2 Behaviour Module

Since migrating to the new framework, the state machine for the behaviour is now integrated directly into the framework. Therefore, behaviours are now written in C++ instead of a special purpose language like XABSL or XTC. These concepts also save a lot of overhead, which was necessary to compile and use XABSL based state machines, like special language parsers, a compiler to create an intermediate code which could then be interpreted by the *XABSL-Engine*, creating a link to the robots' framework and embedding it to the *Robot.exe*. This concept makes it necessary for everything to be known at compile time.
Now, a state machine is compiled to a dynamic link library, as any other module is too, allowing flexibility and fast exchanges of the behaviour, especially during matches for altering whole tactics or adapting behaviours.

Link To Framework

As the state machine needs access to lots of data, which is generated or processed by different modules and afterwards gathered in certain models, a way was needed to access or alter these in a fast and easy manner to keep the state machine's and the robot's data synchronized. This is a time critical part, as it influences the robot's reaction time to alter game or field situations or own states as well. By that, the whole way of playing, which could decide about important advantages is influenced. A problem is the modularity of the new framework and the resulting encapsulated data, which has to be either received or updated.
To keep the state machine simple and reusable, it was decided to use a message passing concept. Any module providing data or being influenced, has to send or receive

15 www2.informatik.hu-berlin.de/ki/XABSL/

messages. During an update phase the current data is exchanged between the modules in every frame of the state machine. Internally, the processable data is then stored in symbols.

General Structure

As mentioned above, all the information coming from or going to the outside of the behaviour module has to be represented internally. For this reason, every accessed module is represented as a model containing symbols within the state machine. The symbols are actually functions, contrary to variables in the old concept, which map to the messages.
For this reason and to keep the idea of a modular concept, every accessed module has to be declared as a separate symbol class. Within these, any data, which is wanted to be read or altered, has to be explicitly registered as a function and then be implemented by mapping the needed message to the symbol function. This has also the advantage of providing some symbols internally and some information additionally by returning preprocessed data from different information sources.
These symbols can then be used in the states to trigger transitions or to initiate actions. The states themselves consist of three parts

- **STATE_INIT**
  The entry point to the state. It can be used to do some preprocessing or initializing local data.
- **STATE_DECISION**
  In this part, all outgoing transitions and their conditions are declared. The transition itself takes effect after executing the third part.
- **STATE_ACTION**
  Within this part, the active reactions are declared, which will then control the robot's actuators in result.

A single state machine is then just built by declaring and implementing the necessary states and switching them together.

Behaviour Structure

As a single state machine, for the whole behaviour, would be much to big and very difficult to keep track of, a behaviour consists of many different state machines. This modular concept also allows to just exchange or adapt partial behaviour without having significant unwanted side effects on other behaviours. It is better to arrange many different state machines in an hierarchical way to achieve a circle free decision tree.
Due to this concept, a top level behaviour as root of the decision tree has to be designed. In every frame this will be the entry point of the complete behaviour state machine and

should therefore contain the state machine consisting of the most important and critical decisions. Beneath this, theoretically infinite hierarchy levels could be arranged. Normally, the second level should be the behaviour, processing the instructions given from the game controller and by this, following the referee's decisions, controlling the match. From the third level downwards, the state machines, which then actually control the team's tactic, are arranged. But most states of the state machines can choose to make decisions and not to invoke any actions of the robots.
This is mostly done at the bottom level, the leaves, of the decision tree. The states here contain some basic actions like walking to a certain direction, kicking or something else. For this reason, some basic behaviours, consisting of simple actions up to small state machines, are implemented, which can then be reused in every state machine without always having to code completely down to the bottom level. But these basic behaviours must not be underestimated. Because of their wide use, they have a significant effect on the whole behaviour and gameplay.
As mentioned above, the whole decision tree is executed in every frame, from the top level behaviour down to a leave invoking an action. This ensures that in every frame the hierarchy is followed and allows faster reaction to altered game situations. The concept itself is pretty similar to other concepts for behaviour state machines such as the formally used XABSL.
So after planning the hierarchy of the decision tree one has to plan the decisions at which the different transitions are followed. This is a bit tricky, as one has to keep in mind that the used information is noisy which makes a hard switching condition hard to define on one hand; and hard switching of behaviours - as well as of states within a single behaviour - should be avoided as it would lead to a clattering-problem as occuring with the use of simple two-position regulators. Therefore it is useful and recommended to have different decision criteria for entering and leaving a behaviour or state which must not overlap but have a certain clearance between them. The clearance's extensiveness influences how fast a decision tree can react to altered values, but also the sensitivity to noises and errors and has to be considered very well. This principle is called state hysteresis.
When implementing a hierarchy or a single behaviour this has to be done manually by the designer, which demands a certain carefulness but gives also the possibility to adapt the state machines to special needs, tactics or improved information through reduced noise, as well as totally different hardware platforms. Therefore this is an important design parameter and can be very powerful if carefully planned.

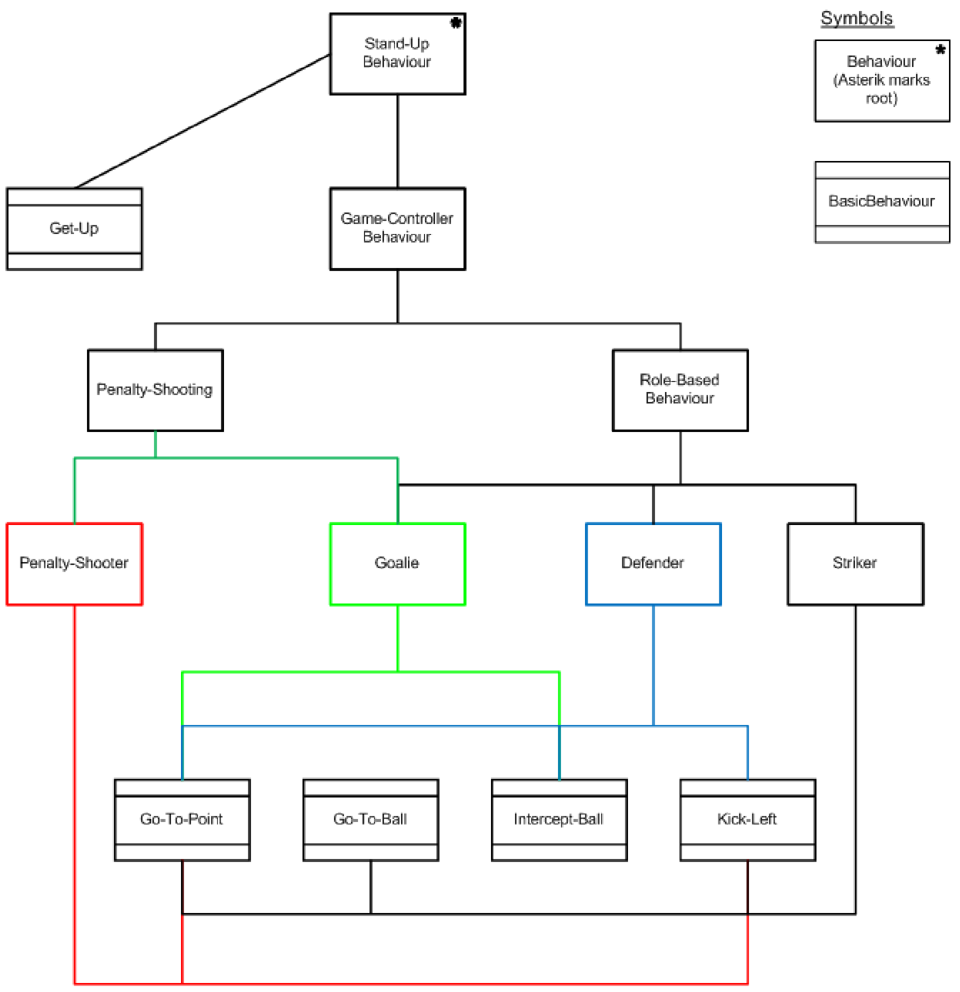


Figure 4.20: Hierarchy Of Example-Behaviour

4.9 Robot Control XP

In this chapter *Robot Control XP* will be presented. In the first part, the key features and mechanisms of *Robot Control* will be discussed, based on the descriptions in [Spr06]. In the second part, the most important problems encountered and changes made to *Robot Control XP* when modifying it towards humanoid robots will be presented. The last part focuses on the network connection, which enables *Robot Control XP* to communicate with the robot's software framework.

4.9.1 Robot Control XP in a Nutshell

Robot Control XP is the third version of a debugging software and part of the debugging architecture, featuring data visualization, data modification, debug switches and remote control.

It was introduced by the *German Team*, an *AIBO* robot soccer team which the university of Dortmund was part of. *RCXP* provides a graphical user interface (GUI) for several debugging mechanisms introduced with the underlying debugging architecture, and thus relieves developers from the task of writing their own complex testing and debugging routines. This lowers the amount of time spent with parameter tuning and other adjustments because there is no need to recompile and restart the robot code for each parameter change.

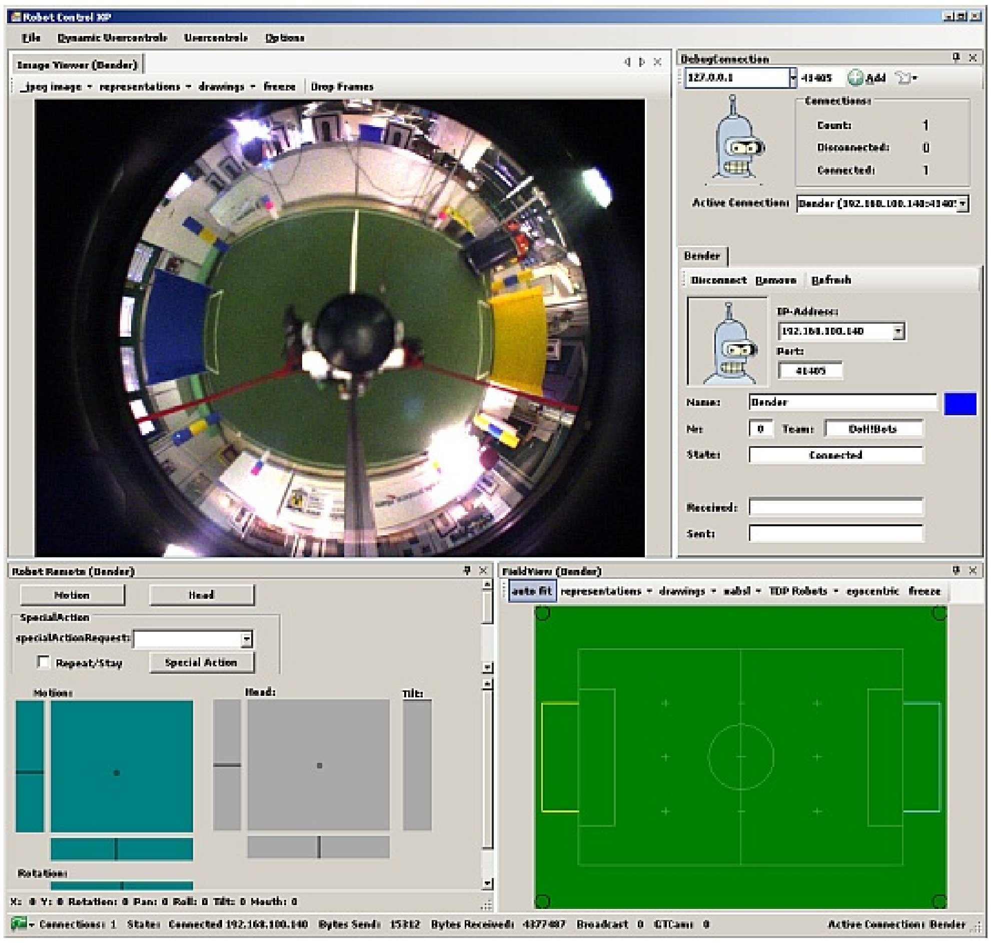


Figure 4.21: Screenshot of Robot Control XP showing some of the available controls

In combination with the old framework the debug switching, visualization and variable modifying was done by using the available debugging macros defined in the framework. The macros can be used everywhere in the robot code. Below some of these macros are listed and their functionality is shortly described.

- DEBUG_RESPONSE(id, code)
  This is the most fundamental debug macro. It switches the given code on or off and is identified by an unique id.
- DECLARE_DEBUGDRAWING(id, type, description)
  This macro declares a debug drawing by specifying a unique id, a type and a description. Possible type values are "drawOnFrontImage", "drawOnOmniImage" and "drawOnField". They define the drawing target as webcam image, omnivision image or virtual field view.
- CIRCLE(id, x, y, radius, penWidth, penStyle, penColor)
  This macro defines an elementary part of the debug drawing which is declared as id. The macro draws a circle with specified radius, pen style, pen width and pen color around the coordinates x, y when the debug drawing is activated. In addition to circles more primitives such as lines or ellipses are available.
- MODIFY(id, object)
  This macro modifies the given object which can either be an instance of a class or a basic variable like an integer. The data is identified by the specified unique id.

The new framework reduces the number of macros because the debugger has issues with following the control flow while using macros. That is why all debugging functionalities are implemented as functions. The old framework's behaviour is emulated, so that the changes to *Robot Control XP* are minimized. Below the functions corresponding to the above macros are listed. To implement a debug response, the return value of a debugResponse function serves as condition for an if statement. The switchable code has to be inserted in the if statement's body.

- bool debugResponse(const char* response)
- void beginDrawing(const char* idName, const char* typeName,
  const char* description)
- void drawCircle(const char* idName, int center_x, int center_y, int radius,
  char penWidth, char penStyle, char penColor)
- template<class T>
  void modify(const char* name, T* t)

Debug responses are the foundation of the debugging system. Nearly all other debugging functionalities make use of them to switch on or off specific code parts. Debug responses can be polled by *Robot Control XP* which means, that *Robot Control XP* sends a poll message and all debug responses answer with a message indicating their existence and availability. In this way *Robot Control XP* is informed about the supported debugging

capabilities.
Debug drawings can be drawn either on a two dimensional representation of the robot soccer field or on one of the robot's camera images. Debug drawings offer an intuitive visualization of data like ball, player and obstacle positions, the field of view or the free angle towards the opponent's goal. This simplifies testing and debugging, for example, of the visual perception algorithms. A wide variety of drawing primitives such as lines, circles, arrows or points is offered for this purpose.
The value of an object or variable made modifiable by a MODIFY macro can be viewed in and modified from within *Robot Control XP*. This functionality is especially useful to try different parameter sets without recompiling or even restarting the code. Furthermore, it is possible to send debug images, e.g. for checking an image filter's effect on the camera image or the actual color table's segmentation qualities. Image visualization is a very important feature because the robots use cameras as their prior sensors.
The old framework is organized in modules, one for each task, e.g. image processing or behaviour control. For each module a variety of interchangeable solutions is possible. The actual solution can be switched at run time. This switching can also be done via *Robot Control XP*'s GUI. The new framework does not support switching of modules at run time.
Besides the possibility to display live data, the received message stream can also be recorded for later analysis. This is, for example, very useful when creating color tables for the robot. Color tables map colors to color classes, so that image areas with specific color ranges can be classified and associated to objects, e.g. the ball. Instead of assigning the color classes while the robot is moved around the field, a set of images can be recorded providing a basis for later color table generation. This proceeding enables the user to navigate through the image sequence including rewinding.

Forming a front end and GUI to the underlying debugging architecture, *Robot Control XP* provides several visualization and modification dialogs:

Standard visualization tasks:

- Image Viewer for Bitmap or JPEG camera images and segmented images. Debug drawings can be shown as overlays.
- Field View for 2D field representation. Displays debug drawings on field positions in a top view.

Special visualization:

- 3D view of the classified color space.
- value-over-time-plot

Control:

- Hardware buttons GUI

- Robot motion remote control

Modification:

- Modification View (Debug Data)

Productive:

- Color Table Tool for color table creation.
- Motion Designer for robot movement design.

The dialogs are handled by managers, where each domain (e.g. image viewing) has its own manager.
Besides the possibility of using *Robot Control XP* with the real robot, it can also be connected to a simulated robot in the simulator as both use the same interface. Thus, it makes no difference for *Robot Control XP* with whom it is communicating.

4.9.2 Modifications in Robot Control XP

As this software was originally designed for the *AIBO* robots, some parts had been altered due to the different needs of humanoid robots and four legged robots. Some changes directly result from the physical differences between the robots, others result from modified or exchanged software. These changes will be pointed out in this chapter.

Hardware motivated modifications

This chapter presents the hardware motivated modifications made.

Besides the change from a four legged robot to a biped robot, the most apparent change is the vision system. It changed from the integrated *AIBO* camera with a resolution of 208 by 160 pixels to an omnivision camera and an auxiliary front camera for kick assistance. The omnivision camera has a resolution of 1024 by 768 pixels. The auxiliary camera is a standard webcam with a resolution of 320 by 240 pixels.

In contrast to the *AIBO*'s camera, which uses YUV color space, both humanoid cameras now natively use the RGB color space. Thus, parts like the color table tool, the color space viewer and the image viewer were modified to work completely in RGB space to prevent unnecessary conversion between the color spaces.
As the omnivision image's size is about 24 times as big as the *AIBO* camera image's size, it was important to speed up image processing. This includes:

- Dropping a rather complex image type, which needed combination of the R, G and B channels for every frame arriving at *Robot Control*
- Speed up image color classification in the color table tool by use of faster classification code: This was achieved by directly reading from the image array.

- Some smaller changes concerning the viewing of big images

The performance problem is to some extent compensated and the GUI is adequately applicable.

In the *AIBO* framework, two different color tables ("128-colortables" = "2 64-colortables") have been used for near and far object classification. As this is not applicable on the humanoid robot's vision concept, all code concerning those 128-colortables has been dropped. Color tables for the front camera and the omnivision camera are saved separately, as this eases combining different color tables for the two cameras.

The Hardware Button Control has been dropped, as the humanoid robot has no software switchable buttons yet. The Remote Control Tool ("ucremote") has been stripped down to the humanoid robots movement directions, this especially means the removal of head movement controls, as the head remains fixed.
A new control, called Angle Control, allows editing every joint's angle with sliders and offers the possibility to save the angle positions to a calibration file. The motivation for calibration and a more detailed depiction of this topic is done in chapter 4.2.5.

Also, there exists a new control for plotting one dimensional data over time (Plotcontrol) and a control visualizing the foot sensor data (Center of pressure and support polygon).

In Chapter 4.4, the development of the motion control is described.

Some minor modifications, especially concerning the visual appearance of Robot Control XP, has been done which will not be discussed here in detail.

Robot's framework motivated modifications

As *Robot Control XP* has not been used for humanoid robots before, the original *Robot Control XP* for the *AIBO* robots was modified to work with the already existent humanoid framework. This especially concerns communication between the robot framework and *Robot Control XP*. For example, the code for the robot connection and receiving of several data types like images and other debug data had to be adapted.

Due to the complete redesign of the robot framework and its evolution in the last months of the project group (see section 4.2), some changes had to be made, again in particular in the communication part. This reaches from messages header changes to a redesign of parts of the debug communication (4.9.3).

Bugfixes

Even though *Robot Control* proved to be a good debugging tool, it still suffers from some bugs. Even though bugfixing covered a big part of the robot control developers work, it will not be discussed here in detail.

4.9.3 Network connection

This chapter gives a brief introduction of the network communication between *Robot Control XP* and the framework. The communication is divided into two parts. The first part is an UDP connection, which broadcasts basic debug informations like the robot's position. The other part is a TCP connection, which can be established to receive more advanced debug informations like pictures and drawings. There are two versions of network code, one for each of the framework versions.

Old framework

The first version of the software framework was based on the framework of team *BreDo-Brothers*, which participated in the *RoboCup* humanoid league in 2006 with a humanoid robot manufactured by *Kondo*. This framework was again based on an rudimentary version of the software framework developed by the *Microsoft Hellhounds* for their four legged *AIBO* robots.
The *BreDoBrothers* framework contained no network code at all. So the first step to run *Robot Control XP* in conjunction with the framework and to use its debug capabilities was to port the network code from a more advanced version of the *Microsoft Hellhounds*' framework to the *BreDoBrothers* framework.
The porting of the network code was straightforward, except for parts using features of the *AIBO* framework, which were not implemented or often not even reasonable in the current framework. In such cases adaptations were necessary.
The framework is structured into three threads performing dedicated tasks. Besides a motion and a cognition thread, the thread performing all network communication is the debug thread. Other threads write their data into a queue and the debug thread sends it over the network. The debug thread is responsible for both network connections. It always broadcasts debug information via UDP and it manages the TCP connection, when *Robot Control XP* is connected to the robot.
Finally, the network communication was running and the debugging could be done as easy as on the *AIBO*s.

New framework

When the implementation of a new software framework for the robots was started, a new implementation of the network code was inevitable. The implementation had to be compatible with *Robot Control XP* and it had to comply with the modular, message based architecture of the new framework.
The network communication in the new framework is handled by an dedicated thread, just like it was done in the old framework. The thread is called "DebugProcess". For

different ways of debugging, for example, a simple text output instead of *Robot Control XP*, different modules can be loaded into this debug thread. The module communicating with *Robot Control XP* is simply called "RobotControlXP". This module directly performs all tasks necessary to communicate with *Robot Control XP* via TCP. If not connected, it listens for incoming connections, otherwise it checks its message queue for debug messages to send over the network, and receives messages to put them into its message queue.
In the new framework the thread, which is responsible for the debug connection to *Robot Control XP*, receives all messages from all other threads because every thread should be able to use the debugging functionalities. To avoid overloading the network, it is not possible to send all data received internally. For example, images of the omnivision camera should only be transferred to *Robot Control XP*, if the user is watching them. Otherwise, huge amounts of bandwidth would be wasted. Therefore, some kind of flag is needed to distinguish between messages, that should be transferred over the network and messages, that should not. It was decided to use a wrapper class called "DebugMessage" to mark network messages. Now, the module simply has to check the message type of its incoming messages and transfer all messages with the message type "ThreadMessage::MsgDebug" over the network.
For debugging purposes *Robot Control XP* supports three fundamental mechanisms, which are introduced in 4.9.1:

- Debug response
- Debug data
- Debug drawing

As mentioned above, debug responses are simple switches, which can enable or disable code blocks. Debug data is a mechanism, which streams variables or instances of classes over the network and enables the user to watch their values. This streaming works bidirectional, so that the user can also change variable values over the network. Debug drawings enable the developer to visualize debug information graphically on images or on a virtual field.
Usages of these mechanisms are spread all over the code, but as there is only one instance of *Robot Control XP* communicating over one connection with the framework, it makes sense to manage these mechanisms centrally. Therefore three singletons were implemented, which basically hold a list of usages of their corresponding mechanism and manage the communication with *Robot Control XP*.
The UDP connection is handled by an actuator running within the debug thread. The actuator is called "RCXPBroadcast". It simply stores an up-to-date version of "RobotPose" and "BallModel" messages and broadcasts their content, which is the current position of the robot and the current estimated position of the ball, over the network.

5 Conclusion and Outlook

5.1 Conclusion

In this project group a humanoid robot was built, which already proved its competitiveness. Even though the DohBots team did not manage to win the preliminary round at the Robocup 2007 in Atlanta, it showed a good performance considering that the DohBots' robots were designed and built within less than a year.
Compared to some other teams the DohBots' robots were quite active on the field. The robot demonstrated good performance in ball recognition at far distances and in motion. Unfortunately, some flaws prevented the team from getting into the quarter finals, in particular an imprecise kick alignment resulting in some unused scoring chances. The main reasons for the imprecise alignment seem to be a quite small opening angle of the front camera and inaccurate odometry.
A lot has been achieved in the project group's year. The group managed to create a completely revised robot body design, which was adapted several times during the last year. Own reliable board designs were created in limited time (e.g. the power board and the carrier board). Sensors were attached to the robot (e.g. the foot sensors and acceleration sensors) and appropriate software was written in order to use these sensors (e.g. the balancing engine). The robot framework was completely redesigned and extended in many points such as the ability to run concurrency on a multicore platform. The use of computer vision with the help of an omnivision camera yields an important advantage over other teams in ball and landmark recognition. A new faster and more flexible robot simulator was introduced and customized. Revised and extended debug possibilities, as well as a gait with spot turn capabilities, are also important improvements.
It can be concluded that a promising platform has been built, which led to a remarkable gain in experience both in hardware and software, and which can serve as a groundwork for subsequent project groups.

5.2 Outlook

The project group had been working on many things such as a new robot body, newly designed electronic parts, using *Microsoft Windows XP* as the robot's operating system for the first time and a partially new designed software framework. As a result of the huge amount of tasks there are some parts that are not completely finished. Some parts have not been used, because there were interdependencies between tasks that could not always be solved in time. Some other parts have been used and are working, but could be enhanced in several ways. For example, the foot sensors development could be continued to make new data sources available for the behaviours (e.g. detection which

foot is standing on the ground) and also the gyroscopes could be integrated.
The time for parameter tuning of the kick alignment was apparently too short. Due to the fact that the DohBots' robots missed most good scoring chances, because of the incorrect alignment and an unpredictable kick, it is recommended to focus more attention on this topics. A better kick alignment and kick could result in a huge leap forward.
To reduce the load for the servos, it should be tried to reduce the weight of the robot. This could also increase the standard kick stability. A slight redesign of the knee joints could make stand-up moves for both sides possible, so that the robot would not have to turn on the back first. The *Microsoft Robotics Studio* does a good job, but has some problems which should be solved in the future, especially the creation of camera images in the simulator itself and the creation of sensor data (e.g. inclination, angular rate and acceleration).

List of Figures

Bibliography

[Cor] CORPORATION, Atmel ; ATMEL CORPORATION (Hrsg.): *ATTINY-26 8-BIT AVR MICROCONTROLLER WITH 2K BYTES IN-SYSTEM PROGRAMMABLE FLASH.* Atmel Corporation, www.atmel.com

[Cor06] CORPORATION, Atmel ; ATMEL CORPORATION (Hrsg.): *8-BIT AVR MICROCONTROLLER WITH 128K BYTES IN-SYSTEM PROGRAMMABLE FLASH.* Atmel Corporation, 2006

[Cza07] CZARNETZKI, Stefan: *Umsetzung eines catadioptrischen Kamerasystems mit Weltmodellierung für einen mobilen autonomen Roboter*, Universität Dortmund, Diplomarbeit, 2007

[Dep] DEPARTMENT, VISHAY S.A. I. ; VISHAY S.A. INTERNATIONAL DEPARTMENT (Hrsg.): *3/4 rectangular multiturn cement trimmer.* VISHAY S.A. International Department

[Dev06] DEVICES, Analog ; ANALOG DEVICES (Hrsg.): *ADM213EARZ EMI/EMC-Compliant RS-232 Line Drivers/Receivers.* Analog Devices, 9 2006

[Gos99] GOSWAMI, Ambarish: Postural stability of biped robots and the foot rotation indicator (FRI) point. In: *International Journal of Robotics Research* 18 (1999), S. 523–533

[Inc] INC., Bothhand E. ; BOTHHAND ENTERPRISE INC. (Hrsg.): *SINGLE RJ45 CONNECTOR MODULE WITH INTEGRATED 10/100 BASE-TX MAGNETICS.* http://www.bothhand.com.tw: Bothhand Enterprise Inc.

[Int] INTERNATIONAL ELECTRONICS ENGENEERING (Hrsg.): *IEE FSR Sensoren, Daten, Eigenschaften und Hinweise zur Handhabung.* International Electronics Engeneering

[KKC04] KONDO KAGAKU CO., LTD. ; KONDO KAGAKU CO.,LTD. (Hrsg.): *Hardware Manual KHR-1.* 1.0. KONDO KAGAKU CO.,LTD., 9 2004

[Kos06] KOSSE, Ralf: *Planung und Implementierung eines evolutionären Ansatzes zur Steuerung eines zweibeinigen Roboters*, Universität Dortmund, Diplomarbeit, 2006

[MIC] MICREL ; MICREL (Hrsg.): *MIC 2026 Dual-Channel Power Distribution Switch.* MICREL

[Mol] MOLEX ; MOLEX (Hrsg.): *Molex 53048 1.25mm Pitch Wire-to-Board Header.* Molex

[Pro99] PRODUCTS, Maxim I. ; MAXIM INTEGRATED PRODUCTS (Hrsg.): *RS-232 Transceivers with AutoShutdown*. Maxim Integrated Products, 1999

[Rec04] RECTIFIER, International ; INTERNATIONAL RECTIFIER (Hrsg.): *IRLR371 HEXFET Power MOSFET*. International Rectifier, 12 2004

[Rec05] RECTIFIER, International ; INTERNATIONAL RECTIFIER (Hrsg.): *IRU3037 8-PIN SYNCHRONOUS PWM CONTROLLER*. International Rectifier, 10 2005

[SEM] SEMTECH ; SEMTECH (Hrsg.): *SR05 RailClamp Low Capacitance TVS Diode Array*. SEMTECH

[Sem97] SEMICONDUCTORS, Philips ; PHILIPS SEMICONDUCTORS (Hrsg.): *PCF8574 REMOTE 8-BIT I/O EXPANDER FOR I2C-BUS*. Philips Semiconductors, 04 1997

[Sem98] SEMICONDUCTORS, Philips ; PHILIPS SEMICONDUCTORS (Hrsg.): *PCF8591 8-BIT A/D AND D/A CONVERTER*. Philips Semiconductors, 07 1998

[Spr06] SPRANGER, Michael: *An Architecture supporting Research and Education in Robotics*, Humboldt-Universität Berlin, Diplomarbeit, 2006