FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Designing an Interface for a Humanoid Robot

Master Work

As a partial satisfaction of the requirements

To obtain the academic degree

Magister/Magistra in Social and Economic Sciences

In

Informatics Management

By:

Jascha Nouri
Student number: 0204119

In the:
Faculty of Informatics at Vienna University of Technology

Advisor: Univ. Prof. Dr.techn. Dr.h.c.mult. Peter Kopacek
Co-Advisor: Dr. Ahmad Byagowi

Vienna, Mai 2010

(Signature of Author) (Signature of Advisor)

Declaration to the Constitution of the work

Jascha Nouri

Hereby, I declare that I have written this work independently, that I am completely specify sources and aids and that I have indicated the bodies of work - including tables, maps and illustrations - and other works, the internet in letter or sense are taken, have given the full names sources.

Vienna, Mai 2010 _____

Abstract

In this work an interface is designed for a human robot. This master work is part of the humanoid robot project called Archie. Archie's project starts from 2004 in institute of handling robotics and technology from Vienna University of Technology. The aim of this project is building a robot who can imitate human's movements like walking.

In this work the control concept of the robot is discussed slightly, later on the interface necessity for such are robot is presented. The main focus of this work is on providing an interface to prepare the necessary data for the robot's joints. Since the whole robot consists of multiple joints connected to each other, calibrating and setting the parameters for each individual joint is a sophisticated phenomena. Therefore this interface is prepared to simply the entire process for the whole robot.

Declaration

I would like to thank my supervisor, Professor Peter Kopacek, for his support and efforts for my diploma thesis. He inspired me to look into the field of mechatronics and humanoid robots. Through his experiences and knowledge I learned a lot.

I would like to thank my student colleague and good friend Lukas Georg, who supported me and designed with me the interface.

In addition, I would like to thank my colleague and good friend Dr. Ahmad Byagowi who has accompanied me in my diploma working hours and always supported me. For his willingness to help me, he motivated me to do my thesis.

Finally, my most profound thanks are to my father, Mag. Mohammed Reza Nouri who is my inspiration model, supported me with his experience, wisdom and goodness in my life. I hope he will be proud of me and hope one day I will be as strong as my father.

I want to thank my mother, Mag. Shahin Nouri who has always given me her love, warmth and strength. I also want to thank my brother Dr. Alireza Nouri who has supported me with his ambition and enforcement capability. He always points the way forward. Also I want to thank my two uncles Shahrokh Assadi and Shahram Assadi, who have supported me in my life ever.

Jascha Nouri

Table of Contents

Problem formulation

The humanoid robot is a sophisticated mechatronic system. A huge number of parameters are required to be set for this machine. For instance, the robot has a minimum of 29 degrees of freedom (DOF). Each degree of freedom is realized by a motor, a gear and a motion controller. On each motion controller, there are several parameters to set for reaching an optimal control. There parameters are to set the torque, velocity and the position sub controllers. All these sub controllers together are used to control the motion of the joint. Since, each one of the joints faces different load properties, setting the joints is a complicated task (i.e., multiple parameters should be set from the joints). The Interface should be designed in a way to provide the requirements for managing the parameters for the robot in a simple way.

1. Introduction

The human is one of the most extraordinary being in the nature. In the evolution from Neanderthal to Homo sapiens, the human being had more developed in instrument, more flexible and more efficient. Humans can be distinguished from animals is different aspects such as the kind of think, to communicate and specially the walking gait. (Burenhult G., 2000)

Over time, the goals of the people get higher and more complex. The design of man is not enough to fulfill certain goals. To meet these goals human needs more tools, machines, technical devices that facilitate the everyday life and make him operations easier and faster.

For example, human uses:

- Cutlery to make them easier to eat food without effort,
- Cars and planes to travel to large, kilometer long distances to go quickly to a target
- Calculators to solve complex mathematical calculations
- And one of the greatest achievements, to accomplish our everyday tasks faster and easier, is the computer.

Computers are now found in all areas of everyday life: They are used in the processing and issuance of information in business agencies, the calculation of statics of structures, the control of machines and automobiles and many other things. Many everyday devices from the mobile phone, clock, technical devices, are now controlled by integrated micro-computers. The most powerful computers are used to simulate complex processes like military tasks, such as the simulation of the use of nuclear weapons. (White R., 2004)

With the increasing performance, we open up new fields of applications such as robots which are programmed to work and support humans.

With the current technical condition, it is now possible to develop robots which derived from the construction, human look, act and move like humans.

Building a human like machine who can imitate human's movements was always a wish for the mankind. History shows, from ancient time, human is looking to fulfill and achieve this dream. Nowadays, the technology is going to give the mankind the ability to start building a human like machine (Humanoid).

This work is as a part of the humanoid robot project, called ARCHIE. ARCHIE is a humanoid robot project which starts at 2004, in Institute of Handling Devices and Robots (IHRT) in Vienna University of Technology (TU Wien), under supervision of Professor Peter Kopacek. Fig. 1 shows a photo from Archie during a press conference. The robot is designed to assist people and imitate human like movements (like human gait). The main goal is to walk with 2 legs (bipedal walking). The human gait is one of the most complicated process in the nature, but the humanoid robot is more flexible and has more possibilities, than other kind of robots, like robots with more than 2 legs or robots with wheels, which they have restricted movements. In this work, the process for designing and implementing an interface for ARCHIE, is presented.

Figure 1: ARCHIE in press conference 2009

2. State of the Art

In this chapter some previous works are presented. Since there are many humanoid robots designed around the world, each one of them uses a different type and structure for their interface. Nonetheless, the aim and the idea behind all of them are based on an identical basis (i.e., all of them are designed for the aim to collect the required information for the robot in an appropriate way). The interface should provide the user a hierarchical structure to ease the process and gives reasonable image of the robot. The complexity of the robot should be simplified for the user to prevent and minimize the user faults in the process of setting the robot.

2.1. Bioloid Interface – Motion Editor

The interface of Bioloid is a graphical interface (Fig. 2). The Motion Editor is a simple design. It consists of a 3D simulation of Bioloid, a list of all the joints and their settings. In the lower area, there are a number of stored positions of the joints.

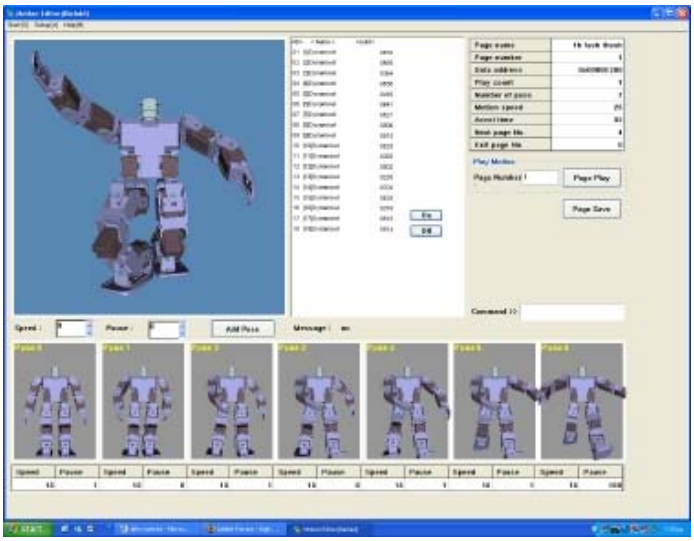

Figure 2: Bioloid – Motion Editor

The velocity and the motion duration time can be assigned. The commands are sent directly via the serial port on Bioloid. In general, the interface is prepared in an optimize shape and has a good overview for optimize usage.

2.2. Interface of Robonova

Similarly to the interface to Bioloid, the Robonova interface is presented (Fig. 3). The interface has a graphic model of Robonova. The individual parameters can be directly entered and stored. There is also the function of the "capture and playback" method. Using the capturing, the controller takes the angle value of all the joints and stores it as a motion.

By using interpolation, the software calculates the required motion between the captured values, in order to prepare smooth and even movement. In general, the interface is compact and has a good overview of all functions.

Figure 3: Robonova Interface (Internet 1, 2010)

2.3. Interface of Acyut

The interface of Acyut is similar to the interface of Robonova (Fig. 4). The central part of the interface consists of a graphical representation of Acyut. The positions of the motors can be set, individually. In the left part of the interface, the motors can be set precisely, for example, the speed, the acceleration on and acceleration time. In the right part,

there are the save settings which stored positions can be loaded and individual positions can be saved.

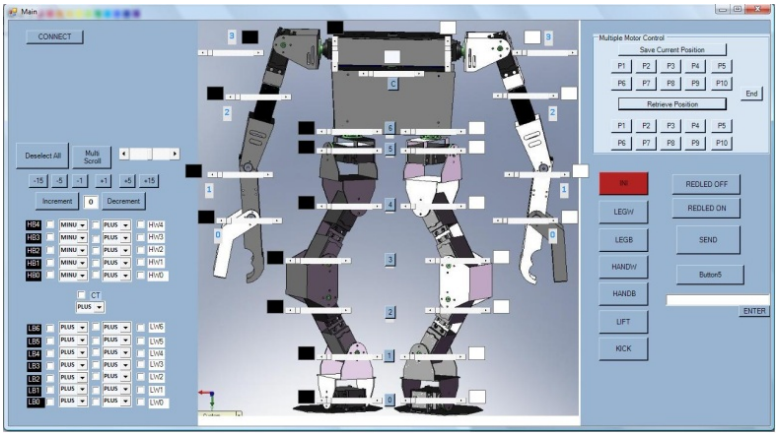

Figure 4: Acyut Interface (Internet 2, 2010)

Also, in this interface, the basic settings have a good overview and enables fast control of the robot.

Most interfaces have similar features and overview, as the presented interfaces. The following properties should include a user-friendly interface:

- Good overview of all adjustable joints
- Flexible input and set of parameters
- Flexible way of storing and loading the set parameters

3. Archie

This chapter presents some briefly descriptions about the robot ARCHIE. Some of the technical specifications of ARCHIE presented in Fig. 5.

Height:	1,2m
Mass:	Approximately 20kg
Operating time:	Min. 60 minutes
Walking Speed:	Min. 0.5 m/s
Degrees of freedom:	Min. 29
On board intelligence	
Dynamic Walking (ZMP)	
Hierarchical, decentralized control structure	
Reasonable low price	

Figure 5: Specifications of ARCHIE

Lower body of Archie is designed and driven using brushless motors. In the joints there are harmonic drives for decreasing the size of the design and increasing the efficiency of the robot. Each joint is control individual and can provide a high performance motion control. All joints are connecting to a central control unit using a customized high speed network.

The following pictures present an overview of ARCHIE. In Fig. 6, a screenshot of the simulation of ARCHIE is shown.

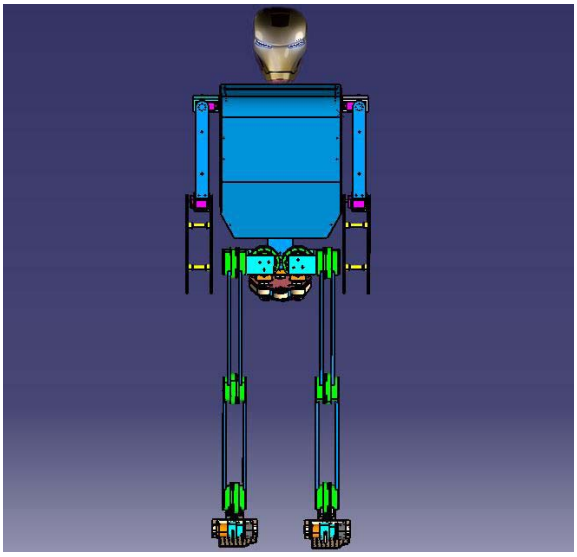

Figure 6: Archie's Mechanical Simulator overview (Byagowi, 2010)

Fig. 7 shows the realization of the design of ARCHIE. The detailed images show the motors of ARCHIE. ARCHIE has 16 motors, 9 of them are brushless motors and the rest brushed motors.

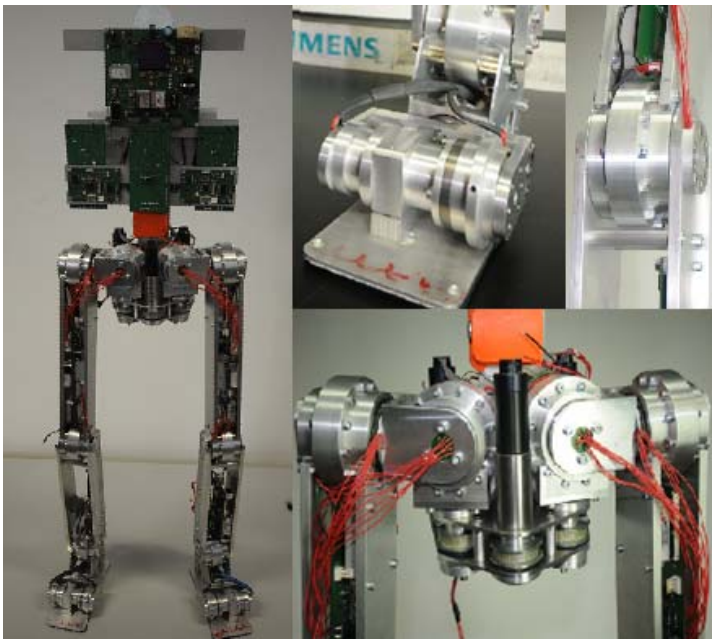

Figure 7: Design of Archie (Byagowi, 2010)

As it is depict in Fig. 7, ARCHIE's design is based on a teen size human shape. The motors move the same way as the human joints.

Fig. 8 shows the prototype of foot mechanism with simulation. The foot mechanism consists of ankle, heel and a toe. The ankle and the toe are for lateral movement, for the foot and the fingers respectively. The heel is for frontal movement.

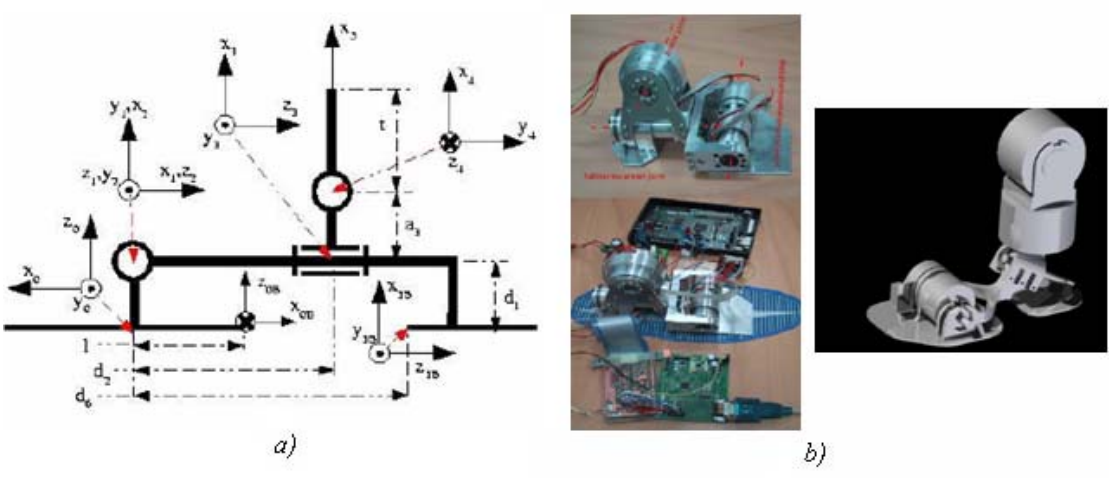

Figure 8: Prototyping of a foot mechanism: a) Denavit Hartenberg (DH) model b) Foot design and simulation (Byagowi 2010)

4. Robot's requirements

4.1. Motion Controller

The motion control is part which belongs to the field mechanics, automation and control. The motion control in robotics is an important phenomenon. The control of a robot is related to the motion controllers (servo). The motion controller handles the position and velocity for each joint by controlling an electric motor as the actuator. The motion controller is one of the most important requirements for the robot. (Tan K. K., 2008)

The architecture of a generic motion controller (Fig. 9):

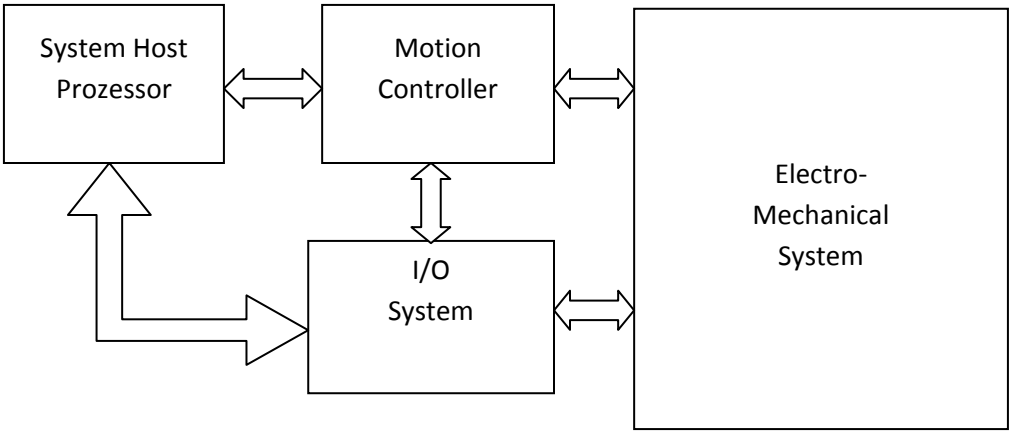

Figure 9: Architecture Motion Controller (Erlendur K., 2009)

Motion controllers are used in many fields, such as in photography, Textile Industry and medicine.

4.2. Motion planner

The motion planner schedules the controller for different velocity and position (as well as torque) based on the time. The motion planner could have the ability to lead the motion controllers of the robot in a way to provide the robot the ability for doing a certain movement. As an instance, the motion planner can provide the robot different type of walking (and rotation) to be used for playing soccer. In a lower level of the work, the motion planner can provide different type of walking based on different speed and walking type.

4.3. Motion controller's hardware

In ARCHIE, the controlling hardware is designed based on distributed computer architecture. In this structure, the joint controllers are connected by a data network to a central controller. The central controller synchronises the joints to prepare combinational movement for the robot (i.e., the central controller commands multiple joints to provide the robot movements which are combinations from different joints an appropriate way).

Each joint is based an individual motion controller. The motion controller is controlling the velocity and position of the motor using a microprocessor.

4.4. Controller

Since the humanoid robot is a non-linear system, the joints of the robot are facing different load properties regarding to the overall pose of the robot. Thus, the control of the robot should be designed in order to provide appropriate performance for a multi-variable system which changes the properties in the variation of time. Besides, the interface of the robot should provide a user-friendly environment to fulfill all the required necessities for tuning the control system of the robot.

4.5. Interface requirements

The interface should provide the requirements for tuning and setting the joints. Since each joint is controlled by a motion controller, the interface should provide appropriate setting ability for the velocity profile and velocity Scheduler to tune each joint properly. In this section there profiles and schedulers are discussed.

4.5.1. Velocity profile

The position movement consists of the phases: acceleration, deceleration and the traversing velocity. The traversing velocity could be based on a constant velocity or coincide with acceleration or deceleration. For the sake of simplicity, in this section the traversing velocity is mind with constant velocity. By this simplification, the joints starts a movement by acceleration; on time of reaching the desired traversing velocity, the joint controller tried to hold the velocity constantly. Once the motion controller sensed the position of the joint close to the desired position (based on the demanded deceleration value), it starts to decrease the velocity of the joint with the demanded rate (based on the desired deceleration). Fig. 10 shows as example for the velocity profile.

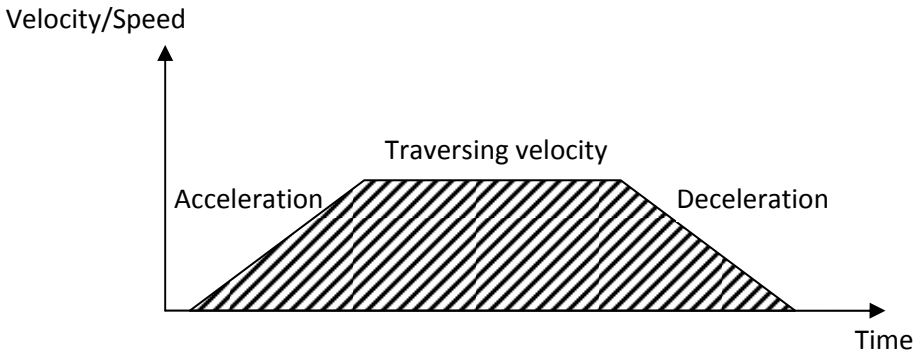

Figure 10: velocity profile consists of acceleration, traversing velocity and deceleration

4.5.2. Torque profile

The torque profile is for scheduling the applied torque to the joint based on the demand of movement. Since the acceleration consists of the 2nd Newton's law of motion, the torque profile should be set on an appropriate value.

By the rotational movement, the mass factor is replaced by the moment of inertia (of the load). The reflected moment of inertia on the joints of a humanoid robot is not stable because it is related to the overall pose of the other joints and links of the robot. The torque is generated by the current of the rotor of the motor. The relation between the generated torque and the current of the motor is represented by the (Kt) value, which is also called the motor constant. The motor current is caused by the applied voltage on the motor form the motion controller. The voltage changes to current by the inverse of the motor's rotor inductance. Fig. 11 shows a block diagram of this consequence.

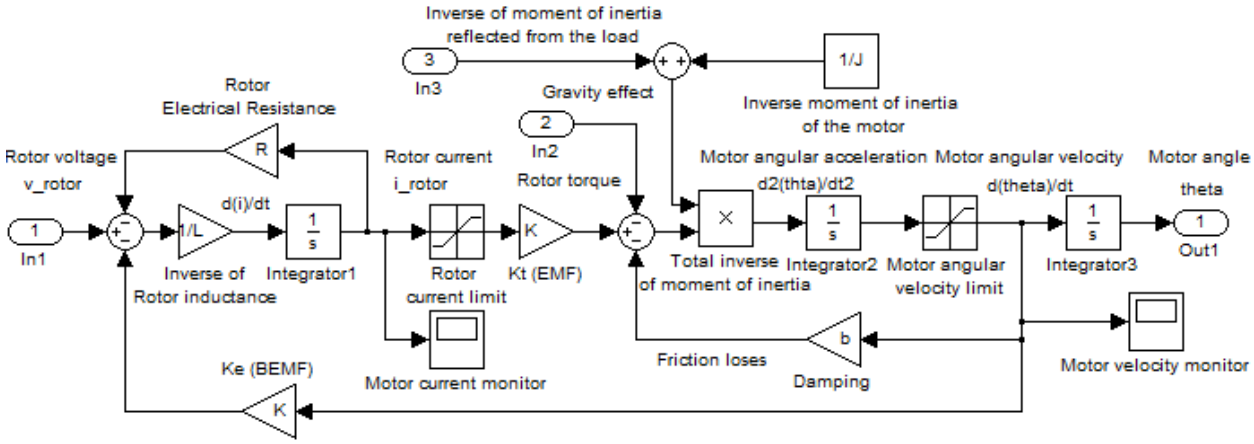

Figure 11: block diagram of the motor's consequence used for the joints of the robot (Byagowi, 2010)

As it is shown in Fig. 11, the motor applied voltage changes to current by the inverse of the motor's rotor inductance. The generated current reduces from the effective terminal voltage with the rotor electrical resistance ratio. In addition the effective voltage of the motor is reduced by the Back Electro Motive Force (BEMF) as it is shown in Fig. 11. The resulted current in motor's rotor provides the output torque. The effective output torque is reduced by the damping factor which is related to the velocity of the robot and the also the reflected gravitational effect on the joint. The effective output torque turns to acceleration by the inverse of the reflected moment of inertia value. Finally using the generated acceleration velocity and position is prepared.

5. Implementation and Functionality

In this chapter, the interface of ARCHIE is presented. The idea was to make a clear and easily adjustable interface for ARCHIE. The interface consists of 3 levels, where in each level, specific sequences, position, acceleration, speed, steps and movements are stored in order to control the functionality of the robot.

5.1. Interface Functionality

5.1.1. Level 1 Functions

In the first level consists of setting the acceleration, acceleration time, speed and torque for each joint. (Fig. 12).

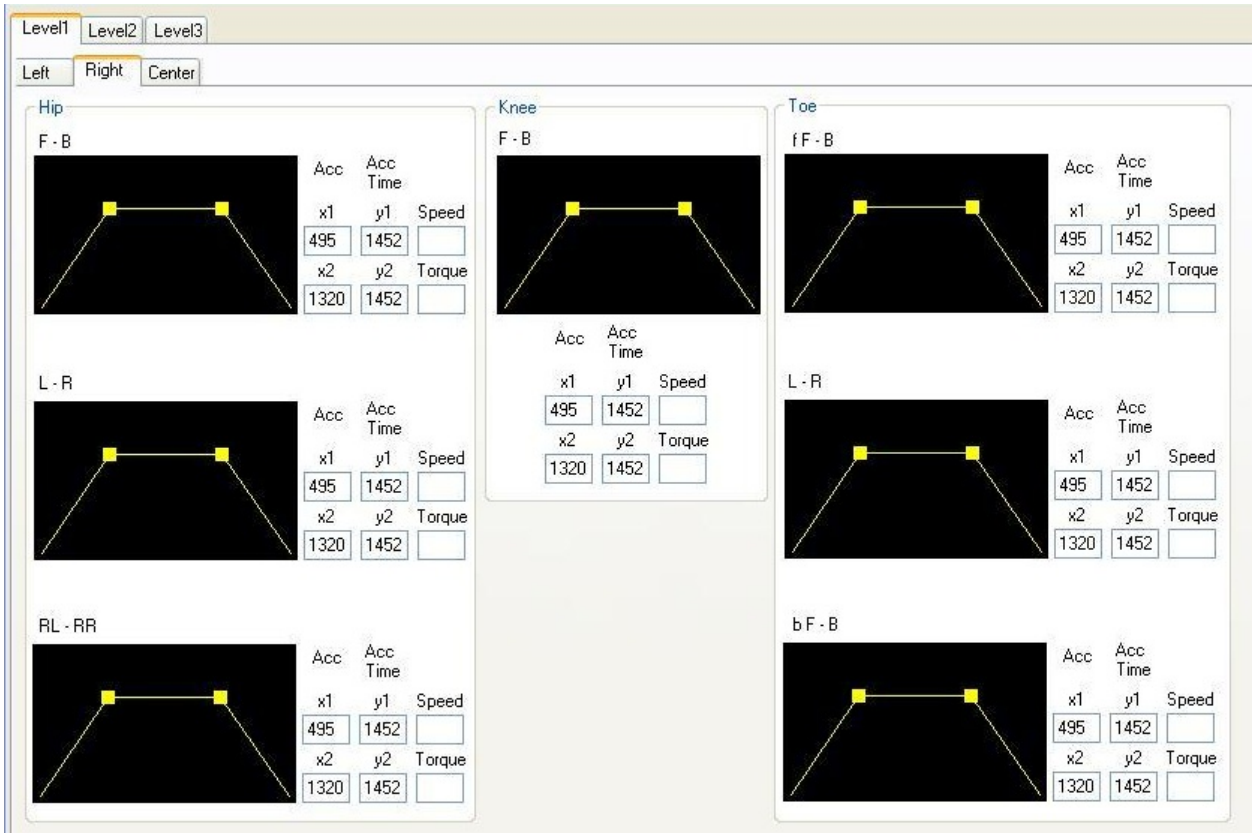

Figure 12: Level 1, Setting the Acc, AccTime, Speed, Torque for the individual joints

By using the interface, the trapezoidal functions can be adjusted via a mouse. ARCHIE has 16 joints (lower body), that means 16 trapezoids are necessary to control the functionality for each joint. The interface stores the properties (Acc, Acc Time, Speed, and Torque), for each individual.

The joints (Fig. 13) are divided into:

- Hip: consist of 8 joints (2 lateral, 3 frontal and 3 transversal)
- Knee: consists of 2 joints (2 lateral)
- Toe: consists of 6 joints (4 lateral und 2 frontal)

The trapezoids are labelled with:

- F-B (Front-Back) = Lateral (L1-L8)
- L-R (Left-Right) = Frontal (F1-F3)
- RL-RR (Rotate Left-Rotate Right) = Transversal (T1-T3)

The trapezoids are subdivided into:

- Left Leg
- Right Leg
- and Center

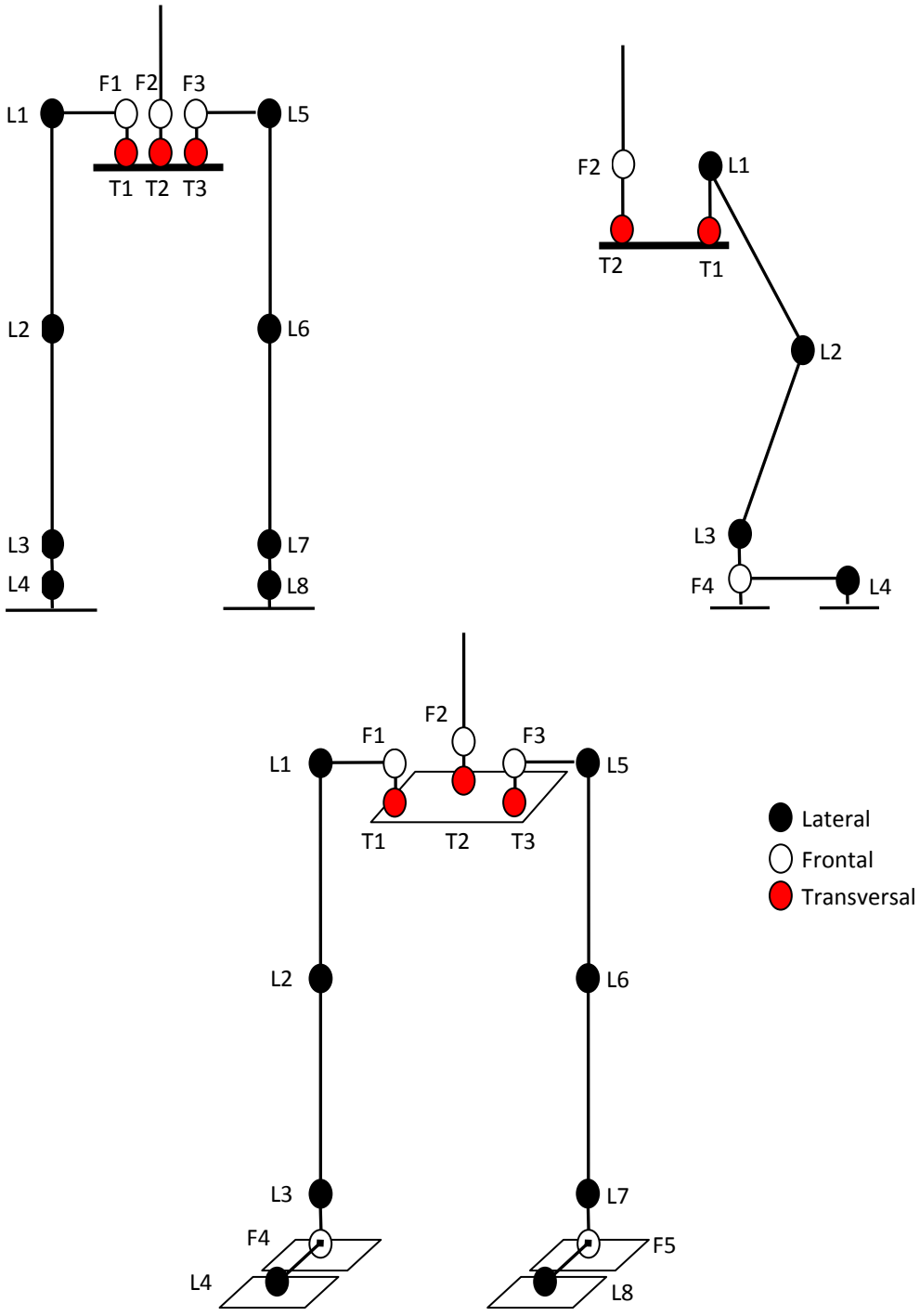

Figure 13: frontal-, lateral-, isometric view and Joints from ARCHIE

First, the properties will be set for each joint (Fig. 14), for example:

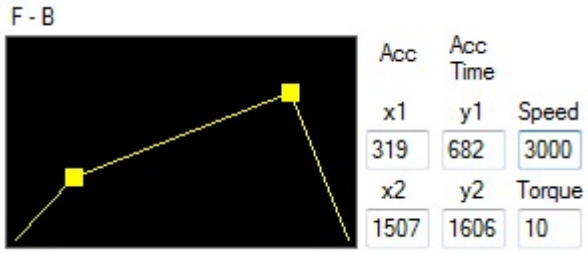

Figure 14: Level 1: Setting the properties

The decision was to use a 2-dimensional trapezoid to adjust the properties, the x axis describes the time and the y axis describes the velocity. The 2 yellow points describes the acceleration and deceleration. Needed for each 16 trapezoid the values are entered.

The values have limitations:

Velocity (Speed): 0-5.000 pulses/s
Acceleration: 0-20.000 pulses/s²
Acceleration Time: maximum torque* moment of inertia (rotational analog of mass)
Torque: 0-20 Nm (Newton meter)

Fig. 15 illustrates the conversion table which is used to translate degree (radiant to the pulses used for the motor):

As an instance when the motor should be moved 45 degree ($\frac{\pi}{4} rad$), the required command (the required pulses) is calculated as follows:

$$\frac{\pi}{4} \times \frac{160 \times 362.25}{2\pi} = \quad 7245\ pulses/s$$

	Brushed DC Motor Based Joint	Brushless DC Motor Based Joint
Gear Ratio	415	160
Encoder, Pulse per Revolution on Motor	2048	362.25
Pulse per each revolution Gear output	415 x 2048	160 x 362.25
Conversion value for radiant	$\frac{415 \times 2048}{2\pi}$	$\frac{160 \times 362.25}{2\pi}$
Conversion value for degree	$\frac{415 \times 2048}{360}$	$\frac{160 \times 362.25}{360}$

Figure 15: Table of Scale

5.1.2. Level 2 Functions

In the second level (Fig. 16), the angle start/end will be set and the steps/motions will be saved (Fig. 18).

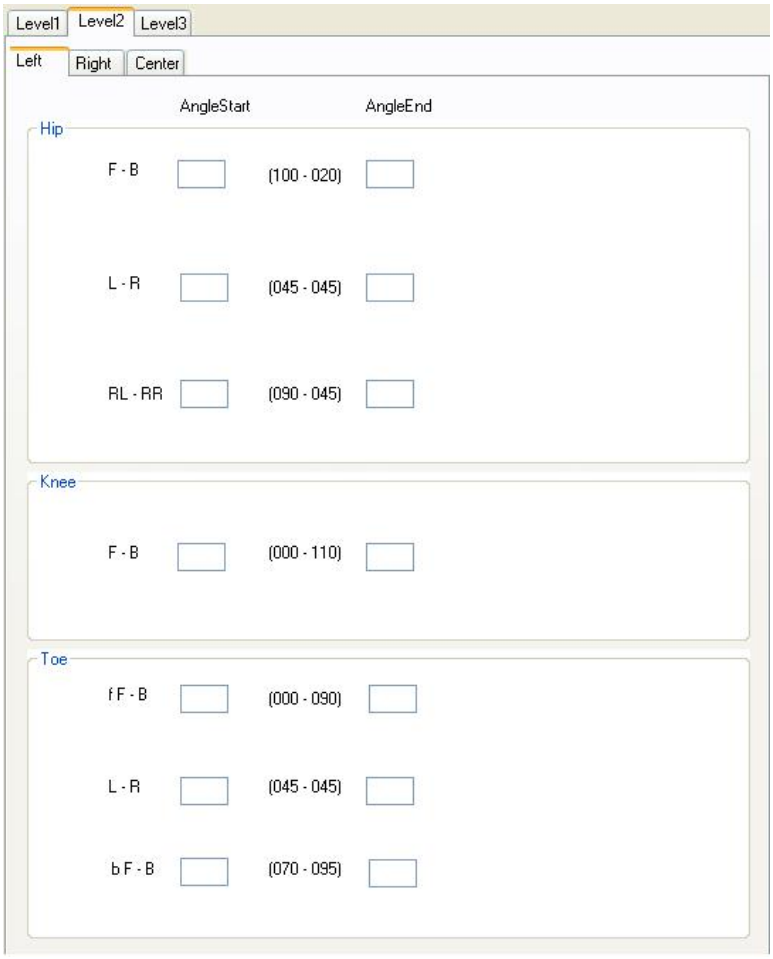

Figure 16: Level 2, Set for Angle Start and Angle End

Similar to Level 1 the angles of each joint are divided into left leg, right leg and center. The distributions of the individual joints to adjust the angles are the same as in Level 1.

First, the angles for all 16 joints will be set. For each joint there is a limitation, which is described between the fields, since otherwise the engines will be overloaded. There are

2 fields that must be fill, the first field "start angle" is entered with the initial state of the joint, and with "stop angle" the end state (Fig. 17).

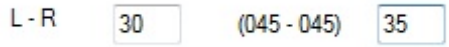

Figure 17: Set the Start/end angles of a joint

After we set the angles for the joint, the next step is to save the settings for Level 1, the settings of the trapezoids and Level 2, the settings of the angles. (Fig. 18)

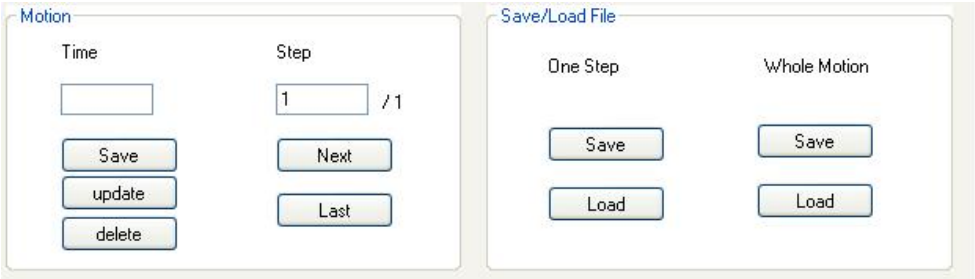

Figure 18: Level 2: Save/Load of Steps and Motions

In the “time” field, the delay time can be filled in, between the steps. One Step is one setting of the trapezoids and angles of all 16 joints. One motion consists of several steps.

There are 2 frames, the motion and the save/load file frame. In the motion frame, it is possible to edit each single step. With the save-button, the step can be saved and after saving it will switch to the next step. The step-field displays, in which field is shown currently and the digit after the field shows how much steps exists. The update-button, updates the step, if any changes are made after one step is saved. With the delete-

button it is possible to delete one step. The next- and last-button are there to switch between the steps, in addition, the next-button also saves the step and switch to the next. The idea with the motion is to save several steps which are required to make just a simple movement, like one gait with ARCHIE.

The save/load file frame is used to save/load single steps and several steps as a whole motion in a XML file, which will save each single setting. (Fig. 19) The “ControlName” describes one field name of the joint it was given in the implementation and “ControlValue” describes the value it was set.

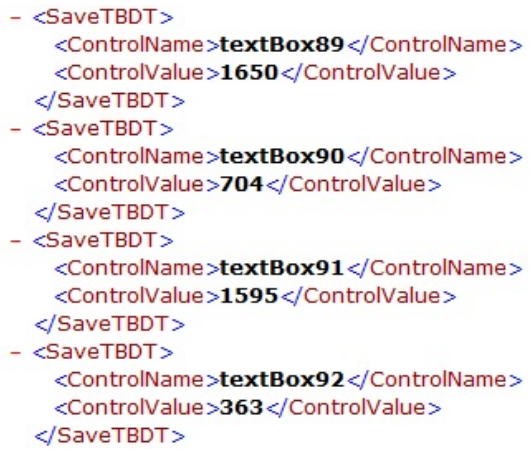

Figure 19: XML File of a step/motion

5.1.3. Level 3 Functions

At last the third level, in this level all steps/motions are listed (Fig. 20) and can be played serially. With the start- and stop- button it is possible to play the current step/motion list. The “SaveList”- and “LoadList”- button is required to save and load the steps or motions, which will be elected to play. This function loads the standard motions which are required, faster, like just make several steps with ARCHIE.

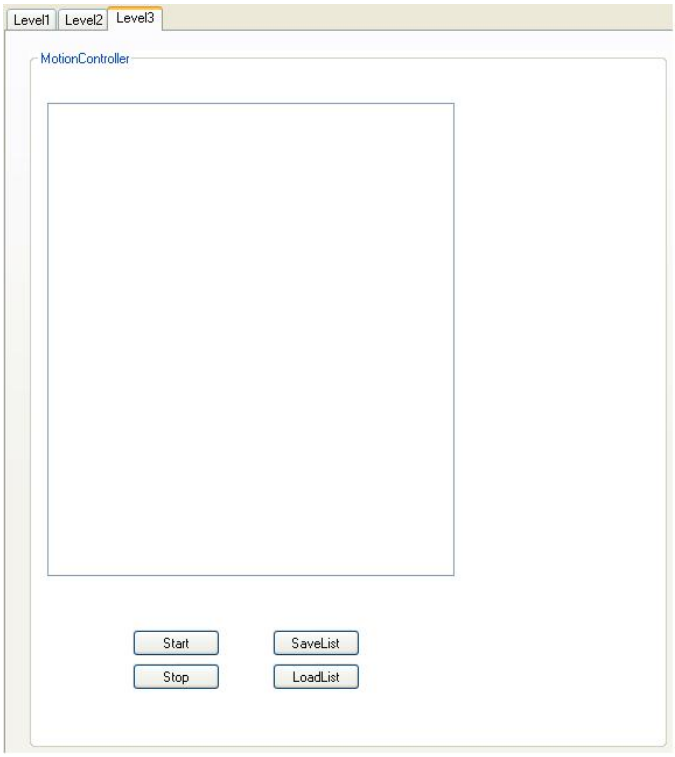

Figure 20: Start/Stop of saved Motions

In addition there is the function of the manual input, if the user wants to test/debugging the interface program (Fig. 21). The user interface consists of manual input of parameters and setting the ports. This part of the interface allows controlling one joint individually.

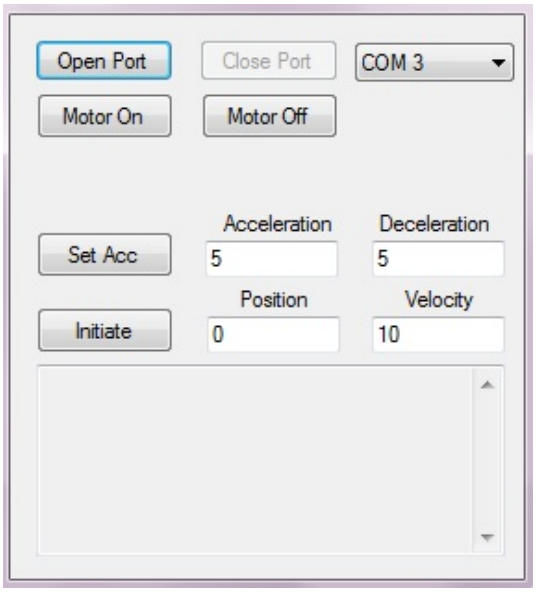

Figure 21: Manual input of parameters

In the first step, it is necessary to set the right port to control a joint, for example “COM3”. If no joint is connected to a port, it cannot be possible to control a joint. For each joint is one port required. The “Open Port” button opens the port which was set. The “Close Port”, which is not available yet, will close the port. The “Close Port” button is only available, if a port is open. After the port is open, it is possible to set the parameters. The first parameters are to turn on/off the motors with the appropriate button (“Motor On”, “Motor Off”). The “Set Acc”-button takes the input parameters of the acceleration and deceleration and sends it to the motor via the serial port. Alternatively, the values have to be entered manually in the text fields. The position field is for setting the position (angle) of the motor, for example, if the value 45 is entered, the motor will rotate 45 degrees. The velocity field is required to enter the speed of the motor. The “Initiate” button sends the given parameters of position and velocity to the port and brings the motor in motion.

For each input parameter, there is a scale (see Fig. 11):

- Acceleration/Deceleration → deg/sec
- Position → deg
- Velocity → deg/sec²

5.2. Interface Implementation

In this chapter, the implementation of the interface is described in detail. The interface was programmed in C# with the help of the program, Microsoft Visual Studio 2005. This program is a complex editor, which can be used for many other projects with other programming languages.

5.2.1. Main Implementation

The main application is presented, which brings the interface start-up. First, it is required to load the interface screen with `Form1_Load`.

```
private void Form1_Load(object sender, EventArgs e)
        {
            // creating a dataset
            myDAL.InitDAL(getTextBoxes4Col(this, null));

            // Focus put on the first PictureBox
            _myPB = pictureBox68;

            // draw all trapezoids
            getPicBoxes(this);
        }
```

The function creates a dataset for saving the values in the textboxes, for saving the set trapezoids and all set configurations with the function `myDAL.InitDAL()`. Also, the focus will be set on the first `PictureBox` and assign it to `_myPB`. `getPicBoxes` draws all trapezoids. Later, there will be a further explanation, what these functions do exactly.

```
private void setTrapezoid(PictureBox pb, Color myColor)
     {
         bmpBack = new Bitmap(pb.Width, pb.Height);
         Graphics.FromImage(bmpBack).Clear(Color.Black);
         pb.Image = (Bitmap)bmpBack.Clone();
```

```
        // Filling the PicturesBoxes with trapezoids
        try
        {
            // node 1
            MarkControl mark1 = new MarkControl(Color.Transparent);
            mark1.Location = new Point(0, pb.Height - 10);
            pb.Controls.Add(mark1);

            // node 2
            MarkControl mark2 = new MarkControl(myColor);
            mark2.Location = new Point((pb.Width - 10) / 3 - 10, 30);
            mark2.Tag = "2";
            pb.Controls.Add(mark2);
```

The function `setTrapezoid` is the main function for creating the trapezoids with the parameters `pb` and `mycolor`. `bmpBack` is used to set the background. A class `System.Drawing.Bitmap` is needed for the operation of images, which are defined by the pixel data. The new point (node 1) is creating a node and for each trapezoid, there are 4 nodes required.

```
            // node 3
            MarkControl mark3 = new MarkControl(myColor);
            mark3.Location = new Point(((pb.Width - 10) / 3) * 2 + 10, 30);
            mark3.Tag = "3";
            pb.Controls.Add(mark3);

            // Default values X+Y from nodes 2 and 3 from the trapezoid are
           entered into the text boxes
            UpdateTextBox(mark2);
            UpdateTextBox(mark3);
            SAVEmark2 = mark2;
            SAVEmark3 = mark3;

            // node 4
            MarkControl mark4 = new MarkControl(Color.Transparent);
            mark4.Location = new Point(pb.Width - 10, pb.Height - 10);
            pb.Controls.Add(mark4);
```

In the function `UpdateTextBox()`, the default values for the node are entered into the text boxes and save to `SAVEmark` for each trapezoid, which are shown in the text boxes.

```
            // All relevant nodes and other values of the trapezoidal object
            passed
             Trapezoid trapezoid = new Trapezoid();
             trapezoid.mark1 = mark1;
             trapezoid.mark2 = mark2;
             trapezoid.mark3 = mark3;
             trapezoid.mark4 = mark4;
             trapezoid.Width = 1;
             trapezoid.Parent = pb;
             trapezoid.Color = myColor;
```

The trapezoid is created with `new Trapezoid()`, the nodes are assigned to this as well as the width and color.

```
                //Events for the node movements add (Node 2)
                mark2.MouseUp += new
System.Windows.Forms.MouseEventHandler(this.Mark_MouseUp);
                mark2.MouseDown += new
System.Windows.Forms.MouseEventHandler(this.Mark_MouseDown);
                mark2.MouseMove += new
System.Windows.Forms.MouseEventHandler(this.Mark_MouseMove);

                // Events for the node movements add (Node 3)
                mark3.MouseUp += new
System.Windows.Forms.MouseEventHandler(this.Mark_MouseUp);
                mark3.MouseDown += new
System.Windows.Forms.MouseEventHandler(this.Mark_MouseDown);
                mark3.MouseMove += new
System.Windows.Forms.MouseEventHandler(this.Mark_MouseMove);

               // adds the newly created trapezoid object in an ArrayList
                Trapezoids.Add(trapezoid);
                Redraw();
            }
```

Mouse events are assigned to the nodes of the trapezoids, to react to drag and drop with `MouseEventHandler()`, which is necessary for mouse event operations. The newly created trapezoid is added to the array of trapezoidal. The `Redraw()` function is used to redraw the trapezoid for updating, which is described in the next chapter in more detail.

5.2.2. Trapezoidal Implementation

First, the implementation of the trapezoids is presented. It is required to draw the trapezoid and the functions name is `DrawTrapezoid()`, which parameters `trapezoid` and `pb` are taken.

```
private void DrawTrapezoid(Trapezoid trapezoid, PictureBox pb)
        {
            Graphics g = null;

            if (trapezoid.Parent == pb)
            {
              g = Graphics.FromImage(pb.Image);
                g.DrawLine(new Pen(trapezoid.Color,
            (float)trapezoid.Width), trapezoid.mark1.Center.X,
            trapezoid.mark1.Center.Y, trapezoid.mark2.Center.X,
            trapezoid.mark2.Center.Y);
                g.DrawLine(new Pen(trapezoid.Color,
            (float)trapezoid.Width), trapezoid.mark2.Center.X,
            trapezoid.mark2.Center.Y, trapezoid.mark3.Center.X,
            trapezoid.mark3.Center.Y);
                g.DrawLine(new Pen(trapezoid.Color,
            (float)trapezoid.Width), trapezoid.mark3.Center.X,
            trapezoid.mark3.Center.Y, trapezoid.mark4.Center.X,
            trapezoid.mark4.Center.Y);
                g.Dispose();
            }
        }
```

In order to draw a trapezoidal figure, the parameters `Trapezoid` and `PictureBox` are taken from the function `DrawTrapezoid()`. The function `DrawTrapezoid()` draws the trapezoidal corresponding to the coordinates. The parent query ensures that the trapezoid is drawn in the correct `PictureBox`. `DrawLine` will draw the line between two points. new Pen initiate a new instance of `System.Drawing.Pen`–class which is required to draw a line.

`trapezoid.mark1.Center.X` and `trapezoid.mark1.Center.Y` gives the coordinates of the points to draw the line. `Dispose()` frees the used ressource from graphics.

```
private void RedrawTrapezoid(Trapezoid trapezoid, Point p, PictureBox pb)
        {
            Graphics.FromImage(pb.Image).DrawImage(bmpBack, 0, 0,
            pb.Image.Width, pb.Image.Height);

            // for each trapezoid of the trapezoidal array
            foreach (Trapezoid g in Trapezoids)
            {
                DrawTrapezoid(g, pb);
            }

            // region which needs to be redrawn
            Region r = getRegionByTrapezoid(trapezoid, p);
            pb.Invalidate(r);
            pb.Update();
        }
```

`RedrawTrapezoid()` redraws the trapezoidal, that the movements of the graphs can be shown. `DrawImage()` draws the specified area of the specified `System.Drawing.Image`, by given size and position. The instruction foreach draws each trapezoidal new in the array. Subsequently, the region is re-refreshed and the `PictureBox` (`pb`) is updated.

```
private void Redraw()
        {
            // for each PictureBox
            foreach (PictureBox myPb in PictureBoxAL)
            {
                  // set background
                if (bmpBack != null)
                    myPb.Image = (Bitmap)bmpBack.Clone();
                else
                {
                  myPb.Image = new Bitmap(myPb.Width, myPb.Height);
                Graphics.FromImage(myPb.Image).Clear(Color.Transparent);
                }
                  // set trapezoidal
                foreach (Trapezoid g in Trapezoids)
                {
                    DrawTrapezoid(g, myPb);
                }
                myPb.Refresh();
            }
        }
```

The function `Redraw()` draws all lines, backgrounds, also trapezoids in the array, new. For each picture box, a background image and a trapezoidal will be set. The corresponding picture box will be refreshed.

```
private void Mark_MouseDown(object sender, MouseEventArgs e)
        {
            this.SuspendLayout();
            isSelected = true;

            // remember coordinates
            _X = e.X;
            _Y = e.Y;
        }
```

The function `Mark_MouseDown()` will be needed for setting the trapezoidal. When a mouse button is pressed, it will set the appropriate variables. X and Y give the coordinates of the mouse while `isSelected` only serves as a flag.

```
private void Mark_MouseMove(object sender, MouseEventArgs e)
     {
            // when a node is moved
         if (isSelected)
         {
                // nodes identify
             MarkControl mc1 = (MarkControl)sender;

                // trapezoidal identify
             Trapezoid g = getTrapezoidByMark(mc1);

                // place of the node to compute
             Point p = new Point(e.X - _X + mc1.Left, e.Y - _Y + mc1.Top);

                // and set
             mc1.Location = p;

                // update values in the text boxes
             UpdateTextBox(mc1);

                // redraw tpezoidal
             RedrawTrapezoid(g, p, _myPB);
          }
      }
```

To bring more flexibility pure for quick and simple settings of the trapezoids, a code was written for use and control over mouse.

When the mouse is moved and a key is pressed (`isSelected`), then the node which is clicked has been identified (`MarkControl`), as the trapezoidal it belongs (`Trapezoid`). Subsequently, the coordinates are reset, the text boxes filled with new values with the function `UpdateTextBox()` and the trapezoidal redrawn with `RedrawTrapezoid()`.

```
private void UpdateTextBox(MarkControl _mc)
        {
            foreach (Control contr in
((Panel)this.Controls.Find(((PictureBox)this.Controls.Find(_mc.Parent.Name,
true)[0]).Parent.Name, true)[0]).Controls)
            {
                if (contr is TextBox)
                {
                    // X – coordinate of the first node to be moved
                    if ((contr.Tag == "x1") && (_mc.Tag == "2"))
                    {
                     contr.Text = Convert.ToString((_mc.Location.X * 11));
                    }
                    // X - coordinate of the second node to be moved
                    if ((contr.Tag == "x2") && (_mc.Tag == "3"))
                    {
                     contr.Text = Convert.ToString((_mc.Location.X * 11));
                    }
                    // Y - coordinate of the first node to be moved
                    if ((contr.Tag == "y1") && (_mc.Tag == "2"))
                    {
                     contr.Text = Convert.ToString(((-1) *
                        (_mc.Location.Y - 94)) * 22 + 44);
                    }
                    // Y - coordinate of the second node to be moved
                    if ((contr.Tag == "y2") && (_mc.Tag == "3"))
                    {
                     contr.Text = Convert.ToString(((-1) *
                        (_mc.Location.Y - 94)) * 22 + 44);
                    }
                }
            }
        }
```

The function `UpdateTextBox()`updates the values in text box. The `foreach` instruction goes through each `PictureBox` and update them in the text box. The coordinates of the text boxes are filled with the current values if the graph will be

changed. Depending on whether it is X- or Y- text box, the values are recalculated. The `if(contr is TextBox)-` instruction includes the both nodes to be moved.

```
private void UpdateGraph(object sender, EventArgs e)
     {
         Trapezoid tp = new Trapezoid();
         try
         {
             foreach (Control contr in ((TextBox)sender).Parent.Controls)
             {
                 // for each picture box
                 if (contr is PictureBox)
                 {
                     tp = getTrapezoidByPicBox((PictureBox)contr);
     // identify the transmitter-textbox and corresponding trapezoid redraw
                     if (((TextBox)sender).Tag == "x1")
                     {
                        Point p = new
Point(Convert.ToInt32(Convert.ToInt32(((TextBox)sender).Text) / 11),
tp.mark2.Location.Y);
                         tp.mark2.Location = p;
                         RedrawTrapezoid(tp, p, (PictureBox)contr);
                     }
     // identify the transmitter-textbox and corresponding trapezoid redraw
                     if (((TextBox)sender).Tag == "x2")
                     {
                         Point p = new
Point(Convert.ToInt32((Convert.ToInt32(((TextBox)sender).Text) + 26)/ 11),
tp.mark3.Location.Y);
                         tp.mark3.Location = p;
                         RedrawTrapezoid(tp, p, (PictureBox)contr);
                     }
     // identify the transmitter-textbox and corresponding trapezoid redraw
                     if (((TextBox)sender).Tag == "y1")
                     {
                         Point p = new Point(tp.mark2.Location.X,
(Convert.ToInt32(((TextBox)sender).Text) * -1) / 22 + 96);
                         tp.mark2.Location = p;
                         RedrawTrapezoid(tp, p, (PictureBox)contr);
                     }
     // identify the transmitter-textbox and corresponding trapezoid redraw
                     if (((TextBox)sender).Tag == "y2")
                     {
                         Point p = new Point(tp.mark3.Location.X,
(Convert.ToInt32(((TextBox)sender).Text) * -1) / 22 + 96);
                         tp.mark3.Location = p;
                         RedrawTrapezoid(tp, p, (PictureBox)contr);
                     }
                 }
             }
         }
     }
```

The function `UpdateGraph()` updates the graph, if a value is entered into the text boxes. Each new point initiates a new instance of `System.Drawing.Point` - class with the given coordinates. There is again calculated separately for X and Y and the trapezoid will be adapted.

```
private void Mark_MouseUp(object sender, MouseEventArgs e)
        {
            isSelected = false;
            ResumeLayout();
            Redraw();
        }
```

`Mark_MouseUp`, release the mouse button pressed down and reset the flag. `ResumeLayout()`, taking the usual layout logic and will force an immediate optional layout for outstanding layout requirements. After that, as usual, everything again will be redrawing.

```
private Trapezoid getTrapezoidByMark(MarkControl m)
        {
            foreach (Trapezoid g in Trapezoids)
            {
                if (g.mark2 == m || g.mark3 == m)
                    return g;
            }
        }
```

`getTrapezoidByMark()`, provides a specific trapezoid object, depending on given node, for each trapezoid.

```
private Trapezoid getTrapezoidByPicBox(PictureBox pb)
        {
            foreach (Trapezoid g in Trapezoids)
            {
                if (g.Parent == pb)
                    return g;
            }
        }
```

`getTrapezoidByPicBox()`, provides a specific trapezoid object, depending on given `PictureBox`, for each trapezoid.

```
private Region getRegionByTrapezoid(Trapezoid g, Point p)
        {
            GraphicsPath gp = new GraphicsPath();
            gp.AddPolygon(new Point[] { g.mark1.Center, g.mark4.Center, p,
            g.mark1.Center });

            RectangleF rf = gp.GetBounds();
            gp.Dispose();

            rf.Inflate(100f, 100f);

            return new Region(rf);
        }
```

`getRegionByTrapezoid()`, provides a specific region depending on given point and trapezoid.

5.2.3. Data Save/load and Action Implementation

One of the most important functions is to store and load the settings of the trapezoid and text boxes. As explained in chapter 5.1.2, the settings are stored in steps, which are saved as an XML file. Furthermore, several steps saved as motion, this serves to re-use of steps and motions for a more flexible and fast adjustment. Now, the Save/Load implementation is described in more detail.

```
        private ArrayList getTextBoxes4Col(Control c, ArrayList l)
        {
            if (l == null) { l = new ArrayList(); }

            foreach (Control contr in c.Controls)
            {
                if (contr is TextBox)
                {
                    l.Add(contr.Name.ToString());
                }
                // Recursion
                if (contr.Controls.Count != 0)
                    getTextBoxes4Col(contr, l);
            }
            return l;
        }
```

The arraylist `getTextBoxes4Col()` is needed as an array list, because it contains about 130 textboxes. The values of all text boxes are stored in a data set, and return this, the whole procedure is done recursively for all the available text boxes of this form.

```
private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                // Values of all text boxes stored in data set
                getTextBoxes(this);
                // Dataset stored in a file
                dlgSaveControls.FileName = "";
                dlgSaveControls.DefaultExt = ".xml";
                dlgSaveControls.Filter = "XML (.xml)|*.xml";
                if (dlgSaveControls.ShowDialog() == DialogResult.OK)
                {
                    string filename = dlgSaveControls.FileName;
                    mySaveDS.WriteXml(filename);
                }
            }
```

The function `button1_click` is the Save-button for saving steps. The values are required from all text boxes with `getTextBoxes()`.

`dlgSaveControls.FileName` gets a string that contains the file name selected in the file dialog box or sets.

`dlgSaveControls.DefaultExt` gets the default file extension or sets.

`dlgSaveControls.Filter` gets the current filter string for file name, which is defined in the dialog box "Save as" or "file type" displayed selection or sets.

`WriteXML()` writes, in the given `System.Data.XMLWriteMode`, the current data, and in need the schematic for the `System.Data.DataSet` specified in the file.

```
private void getTextBoxes(Control c)
        {
            foreach (Control contr in c.Controls)
            {
                if (contr is TextBox)
                {
                  SaveDS.SaveTBDTRow mySaveDSrow =
                  mySaveDS.SaveTBDT.NewSaveTBDTRow();

                    // write control name
                    mySaveDSrow.ControlName = contr.Name.ToString();

                    // write control value
                    mySaveDSrow.ControlValue = contr.Text.ToString().Trim();

                    // add data set
                    mySaveDS.SaveTBDT.AddSaveTBDTRow(mySaveDSrow);
                }

                // Recursion
                if (contr.Controls.Count != 0)
                    getTextBoxes(contr);
            }
        }
```

The function `getTextBoxes()` is storing values of all text boxes in a data set as a step. In this data set, the control name and the control text will be written. `mySaveDSrow()` stores the data in the record. Recursively, all text boxes will be collected with `getTextBoxes(contr)`.

For storing the values and textboxes, it is the same procedure, except that is required an array list for several steps.

```
private void setTextBoxes4Motion(Control c, DataRow dr)
        {
            foreach (Control contr in c.Controls)
            {
                if (contr is TextBox)
                {
                    contr.Text = dr[contr.Name.ToString()].ToString();
                    UpdateGraph(contr, null);
                }

                // recursion
                if (contr.Controls.Count != 0)
                    setTextBoxes4Motion(contr, dr);
            }
        }
```

The function `setTextBoxes4Motion()` is loading values of all text boxes from a data set. The parameter `dr` is needed for loading the text in the text boxes, in addition, it is required to update the trapezoids with the function `UpdateGraph()`. As in the storing procedure, it runs again recursively.

```
private void resetTextBoxes(Control c)
        {
            foreach (Control contr in c.Controls)
            {
                if (contr is TextBox)
                {
                    if (contr.Tag == "x1") { contr.Text = "495"; }
                    else if (contr.Tag == "x2") { contr.Text = "1320"; }
                    else if ((contr.Tag == "y1") || (contr.Tag == "y2")) {
contr.Text = "1452"; }
                    else { contr.Text = ""; }
                }

                // recursion
                if (contr.Controls.Count != 0)
                    resetTextBoxes(contr);
            }
        }
```

The function `resetTextBoxes()`, will be reset the text boxes will to default values, so that the trapezoid is always properly placed back to its initial state. And there, the procedure runs recursively.

```
private void getPicBoxes(Control c)
        {
            foreach (Control contr in c.Controls)
            {
                if (contr is PictureBox)
                {
                    // add to array list
                    PictureBoxAL.Add(contr);

                    // create trapezoid
                    setTrapeziod((PictureBox)contr, Color.Yellow);
                }

                // recursion
                if (contr.Controls.Count != 0)
                    getPicBoxes(contr);
            }
        }
```

`getPicBoxes()` goes thru all the existing picture boxes by a form (recursively) and generates the corresponding trapezoids. At this point, the line color of the trapezoids will be specified.

```
private void button3_Click(object sender, EventArgs e)
        {
            try
            {
                dlgOpenControls.FileName = "";
                dlgOpenControls.DefaultExt = ".xml";
                dlgOpenControls.Filter = "XML (.xml)|*.xml";

                if (dlgOpenControls.ShowDialog() == DialogResult.OK)
                {
                    string filename = dlgOpenControls.FileName;

                    mySaveDS.ReadXml(filename);
```

```
                    // for each data set
                foreach (SaveDS.SaveTBDTRow myLoadDSrow in mySaveDS.SaveTBDT)
                    {

      ((TextBox)Controls.Find(myLoadDSrow.ControlName, true)[0]).Text =
      myLoadDSrow.ControlValue;

                        // search the corresponding text box control
                        if (myLoadDSrow.ControlName.Contains("362"))
                        {
                            // set the values

GetPicBoxFromTextBox(((TextBox)Controls.Find(myLoadDSrow.ControlName,
true)[0]));
                        }
                    }
                }
            }
```

The function `button3_Click()` loads a step and fill the text boxes. Also, the trapezoids will be adjusted. `dlgOpenControls` opens the standard dialog box for loading the step. `mySaveDS.ReadXML("filename")`, loads the XML file into the data set. All values that are saved in the XML-file, will be entered in the corresponding text box.

```
private void jumpToStep(string _name)
        {
            try
            {
                setTextBoxes4Motion(this, myDAL.ReadStep(_name));
            }
            catch (Exception ex)
            {
                MessageBox.Show("Step dosen't exist");
            }
        }
```

The function `jumpToStep()`, like the name said is to jump to a step. By pressing the enter key, it will load the data into the text box.

```
        private void MyKeyUp(object sender, KeyEventArgs e)
        {
            try
            {

                string tmp = ((TextBox)sender).Tag.ToString();
                if ((tmp.Contains("x")) && ((((TextBox)sender).Text) != ""))
                    if (Convert.ToInt32(((TextBox)sender).Text) > 1800)
                    {
                        ((TextBox)sender).Text = "1800";
                        e.Handled = true;
                    }
                if ((tmp.Contains("y")) && ((((TextBox)sender).Text) != ""))
                    if (Convert.ToInt32(((TextBox)sender).Text) > 2000)
                    {
                        ((TextBox)sender).Text = "2000";
                        e.Handled = true;
                    }
```

In function `MyKeyUp()`, the maximum value of the text boxes are set, depending on what may stand in it. `tmp.Contains()`, returns a value indicating whether the specified `System.String` - object is present in this string. `Convert.ToInt32()`, converts the string representation of a number, with respect to a specified base in the corresponding 32-bit signed integer. For the other text boxes, it is required the same code with their maximum value.

```
private void btnNextStep_Click(object sender, EventArgs e)
        {
            int intAktuell = Convert.ToInt32(txtStep.Text.Trim()) + 1;
            int intMaximum = 0;
            string guid = "";

            if (myDAL.ReadStep(txtStep.Text.Trim()) == null)
            {
                myDAL.WriteStep(getTextBoxes4Motion(this, null));
            }
```

The function `btnNextStep_Click()`, is for the “Next Step” - button, which is required to save the actual step and go to the next. If an actual data set is not available, it will be saved as a new step.

```
else
      {
      myDAL.UpdateStep(getTextBoxes4Motion(this, null), txtStep.Text.Trim());
      }
```

Otherwise, the step will be updated with the function, `UpdateStep()` . `getTextBoxes4Motion()`, values of all text boxes stored in a data set. Recursively, the text boxes will be captured, for the motion.

```
            DataRow dr =
myDAL.ReadStep(Convert.ToString(Convert.ToInt32(txtStep.Text.Trim()) + 1));
            if (dr == null)
            {
                resetTextBoxes(this);
                intMaximum = myDAL.LastID() + 1;
            }
```

Here, it is required an if-instruction to see if the next data set is available. If there isn't any next data set, the function `resetTextBoxes()` reset the text boxes to default values, so that the trapezoid is always reset correctly in the initial state.

```
            else
            {
                setTextBoxes4Motion(this, dr);
                intMaximum = myDAL.LastID();
                guid = dr[0].ToString();
            }
```

Otherwise, the `setTextBoxes4Motion()` load values of all text boxes from a data set (recursive). `LastID()`, returns the maximum entries. For the last step-button it is required the function `btnLastStep_Click()` and the same code like `btnNextStep_Click()`.

```
private void btnStepSave_Click(object sender, EventArgs e)
        {
            // data send to DAL
            myDAL.WriteStep(getTextBoxes4Motion(this, null));

            // reset text boxes
            resetTextBoxes(this);
        }
```

The function `btnStepSave_Click()` is for the “Save” - button to simply save one step. The data will be sending to our data array list (DAL) to save the entries of the text boxes. The `resetTextBoxes()`, reset text boxes, for setting the next step.

```
private void button7_Click(object sender, EventArgs e)
        {
            // data send to DAL
            myDAL.UpdateStep(getTextBoxes4Motion(this, null), "1");
        }
```

It is required to update the step when the "next" button is used, since changes can be made. The data will be sending to DAL.

```
private void button8_Click(object sender, EventArgs e)
        {
            int intAktuell = 0;
            int intMaximum = 0;
            string guid = "";

            try
            {
                // data send to DAL
                myDAL.DeleteStep(txtStep.Text.Trim());
                intMaximum = myDAL.LastID();

                if (Convert.ToInt32(txtStep.Text.Trim()) <= myDAL.LastID())
                {
                    DataRow dr = myDAL.ReadStep(txtStep.Text.Trim());
                    setTextBoxes4Motion(this, dr);
                    intAktuell = Convert.ToInt32(txtStep.Text.Trim()) - 1;
                    guid = dr[0].ToString();
                }
```

```
            else
            {
                intAktuell = Convert.ToInt32(txtStep.Text.Trim()) - 1;
                DataRow dr = myDAL.ReadStep(intAktuell.ToString());
                setTextBoxes4Motion(this, dr);
                guid = dr[0].ToString();
            }
        }
```

The `button8_Click() function` is for deleting the step and again, the data will be send to DAL. An if-instruction is required, if it is the last step of a motion. The last step of a motion is simply subtracted by 1. If it is not the last step, it also subtracted by 1, as are the next steps.

```
    private void button6_Click(object sender, EventArgs e)
    {
        try
        {
            // data set save in a file
            dlgSaveControls.FileName = "";
            dlgSaveControls.DefaultExt = ".xml";
            dlgSaveControls.Filter = "XML (.xml)|*.xml";
            if (dlgSaveControls.ShowDialog() == DialogResult.OK)
            {
                string filename = dlgSaveControls.FileName;
                myDAL.WriteXML(filename);
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
```

This function `button6_Click()` is for saving a motion. As it is required the storage for the step, it is also required the storage of the motion. As a motion consists of several steps, the implementation is not much different than to save steps. The motions also will be saved as an XML file. Again, it is required the `dlgSaveControls` for the standard windows dialog box. The XML file will be written in DAL.

```
        private void button5_Click(object sender, EventArgs e)
        {
            try
            {
                dlgOpenControls.FileName = "";
                dlgOpenControls.DefaultExt = ".xml";
                dlgOpenControls.Filter = "XML (.xml)|*.xml";

                if (dlgOpenControls.ShowDialog() == DialogResult.OK)
                {
                    string filename = dlgOpenControls.FileName;
                    myDAL.ReadXML(filename);

                    DataRow dr = myDAL.ReadStep("1");
                    if (dr == null)
                    {
                        // No -> New file
                        resetTextBoxes(this);
                    }
                        else
                    {
                        // Yes -> load
                        setTextBoxes4Motion(this, dr);
                        SetStepInt(1, myDAL.LastID(), dr[0].ToString());
                    }
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
```

The function `button5_Click()`, is responsible for loading the motion. The code is similar to the motion save. It is required to check, if the `DataRow` is empty or not. So, if the `DataRow` is empty the text boxes will be reset, otherwise the set boxes will be loaded.

5.2.4. Data Array List (DAL)

In this chapter, the Data Array List (DAL) is presented in more detail. The DAL is needed to save the settings of the steps and motions in an array list.

```
public DataSet InitDAL(ArrayList _TextBoxen)
        {
            // Tables for generating data set
            DataTable myStepDT = new DataTable();
            myMotionDS.Tables.Add(myStepDT);

            // Columns to create, for the tables
            myMotionDS.Tables[0].Columns.Add("GUID");
            myMotionDS.Tables[0].Columns.Add("ID");

            // Columns to create, for the text boxes
            for (int i = 0; i < _TextBoxen.Count; i++)
            {
                myMotionDS.Tables[0].Columns.Add(_TextBoxen[i].ToString());
            }
            // Set columns features
            myMotionDS.Tables[0].Columns[0].Unique = true;
            return myMotionDS;
        }
```

For the first step, the Data Array List (DAL) should be initiated, where the tables are created. With new `DataTable()`, a new instance of `System.Data.DataTable` – class is initialized without arguments. The generated data set (`new DataSet()`) are assigned to `myMotionDS`. The command `myMotionDS.Tables.Add()`, creates a new `System.Data.DataTable` - object with a default name and adds it to the list. The next step, the columns of the tables and text boxes will be created. For creating the columns of the text boxes, we require a loop. A condition which has to be met must be the uniqueness of all the values in the rows of columns (`Unique`).

```
        public Boolean WriteStep(ArrayList _TextBoxen)
        {
            Boolean result = true;
            DataRow myRow = myMotionDS.Tables[0].NewRow();

            try
            {
                myRow[0] = System.Guid.NewGuid().ToString();
```

The function `WriteStep()`, saves a step in the dataset. `NewRow()`, creates a new `System.Data.DataRow` with the scheme of the tables.

```
if (myMotionDS.Tables[0].Rows.Count > 0)
                {
                    int tmp =
Convert.ToInt32(myMotionDS.Tables[0].Rows[myMotionDS.Tables[0].Rows.Count -
1][1]);
                    myRow[1] = tmp + 1;
                }
                else
                {
                    myRow[1] = 1;
                }
```

The instruction retrieves the total number of `System.Data.DataRow` - objects from the list. Each row gets an ID, the else-instruction is for the beginning ID.

```
for (int i = 2; i < _TextBoxen.Count + 2; i++)
                {
                    myRow[i] = _TextBoxen[i - 2].ToString();
                }

                myMotionDS.Tables[0].Rows.Add(myRow);

            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
                result = false;
            }

            return result;
        }
```

This loop will paste the values of the text boxes. `Add(myRow)` creates a row with the specified values and adds it to the `System.Data.DataRowCollection`.

`Tables[0]` gets the collection of tables contained in the `System.Data.DataSet`. At the end the result of `WriteStep()` will be returned.

```
public Boolean UpdateStep(ArrayList _TextBoxen, string _name)
        {
            Boolean result = true;
            try
            {
                // search data row
                  DataRow myRow = myMotionDS.Tables[0].Select("ID = '" +
                  Convert.ToInt32(_name) + "'")[0];

                // paste the values of the text boxes
                for (int i = 2; i < _TextBoxen.Count + 2; i++)
                {
                    myRow[i] = _TextBoxen[i - 2].ToString();
                }

                myRow.AcceptChanges();
            }

            catch (Exception ex)
            {
                result = false;
            }

            return result;
        }
```

The function `UpdateStep()` will make an update of the step if any changes will be made. `myMotionDS.Tables[0].Select()` gets an array of all `System.Data.DataRow` - objects that meet the filter criteria in the sort order and fit the specified state. The function `AcceptChanges()` performs a "commit" for all the changes that have been made in this row, since the last call of `System.Data.DataRow.AcceptChanges()`.

```
public DataRow ReadStep(string _name)
        {
            DataRow myRow = null;
            try
            {
                // search data set
            myRow = myMotionDS.Tables[0].Select("ID = '" +
            Convert.ToInt32(_name) + "'")[0];
            }
            catch (Exception ex)
            {

            }
            return myRow;
        }
```

The `ReadStep()` function, loads the saved step, it works like `UpdateStep()`, except that only the values in data set will be loaded in the text boxes.

```
public Boolean DeleteStep(string _name)
        {
            Boolean result = true;
            int intTmp = Convert.ToInt32(_name);

            try
            {
                // search data row
                  DataRow myRow = myMotionDS.Tables[0].Select("ID = '" +
                  Convert.ToInt32(_name) + "'")[0];

                // delete data row
                myMotionDS.Tables[0].Rows.Remove(myRow);

                // correct IDs/Names
                for (int i = 0; i < myMotionDS.Tables[0].Rows.Count; i++)
                {
                    myMotionDS.Tables[0].Rows[i][1] = i + 1;
                }
            }
            catch (Exception ex)
            {
                result = false;
            }
            myMotionDS.AcceptChanges();
            return result;
        }
```

The function `DeleteStep()`, deletes a step which is processed, it works like `WriteStep()` and `ReadStep()`. `Remove(myRow)` removes the specified `System.Data.DataRow` from the collection. The loop will correct the ID's and names of the data sets.

```
public Boolean WriteXML(string _path)
        {
            Boolean result = true;
            try
            {
                myMotionDS.Tables[0].WriteXml(@_path);
            }
            catch(Exception ex)
            {
                MessageBox.Show(ex.Message);
                result = false;
            }
            return result;
        }
```

The `WriteXML()` function creates a XML-File for saving steps or motions. `WriteXml(@_path)` writes the specified file and the specified `System.Data.XmlWriteMode` in the current data and requires the schema for `System.Data.DataTable`.

```
        public Boolean ReadXML(string _path)
        {
            Boolean result = true;
            try
            {
                myMotionDS.Tables[0].ReadXml(@_path);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
                result = false;
            }
            return result;
        }
```

The function `ReadXML()` is required to load the saved XML file. The code works like the `WriteXML()` function and at the end, the results will be returned.

5.2.5. Implementation of Debugging Console

In this chapter the implementation of the debugging console will be presented, which is used in LVL3 for manual input parameters for each joint.

```
private void buttonStart_Click(object sender, EventArgs e)
        {
            serialPort1.BaudRate = 19200;

            serialPort1.Open();
            if (serialPort1.IsOpen)
            {
            buttonStart.Enabled = false;
            buttonStop.Enabled = true;
            }

        }
```

The function `buttonStart_Click()` is required to open the ports for sending the entered parameters. `serialPort1.Baudrate` sets the serial baud rate. In this case, the velocity is 19200 bit/sec. `serialPort1.Open()` opens a new serial port connection. The if-instruction will check whether the port is open or not. If the port is open, the “Open Port”-button will be inactive and the “Close Port”-button will be active.

```
        private void buttonStop_Click(object sender, EventArgs e)
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Close();
                buttonStart.Enabled = true;
                buttonStop.Enabled = false;
            }

        }
```

Similar to `buttonStart_Click()`, the function `buttonStop_Click()` will close the opened port. The if-instruction checks if the port is open and close it. The “Open Port”-button will be active and the “Close Port”-button will be inactive.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            if (serialPort1.IsOpen) serialPort1.Close();
        }
```

Likewise the function `buttonStop_Click()`, it checks if the port is open or not. This additional function is required if the program will be exit to disconnect the connection, in case an open port is existing.

```
private void DisplayText(object sender, EventArgs e)
        {
            textBox1.AppendText(RxString);
        }
```

The function `DisplayText()` is required to add text to the current text in a text box. `RxString` is a variable of type string, which saves the entered text for displaying in the text box.

```
private void serialPort1_DataReceived(object sender,
System.IO.Ports.SerialDataReceivedEventArgs e)
        {
            RxString = serialPort1.ReadExisting();
            this.Invoke(new EventHandler(DisplayText));
        }
```

The function `serialPort1_DataReceived()` is required to save the entered parameters to `RxString` and display it into the text box. `ReadExisting()` return all immediately available bytes in both the stream and the input buffer.

```
private void button6_Click(object sender, EventArgs e)
        {
           string eingabe = textBox2.Text;
           int c = Convert.ToInt32(eingabe);
           c = (int)c * (160 * 360) / 360;
           string tempac = Convert.ToString(c);
```

```
            serialPort1.Write("AC=" + tempac + "\n\r");
            string eingabe1 = textBox4.Text;

            int d = Convert.ToInt32(eingabe1);
            d = (int)d * (160 * 360) / 360;

            string tempdc = Convert.ToString(d);
            serialPort1.Write("DC=" + tempdc + "\n\r");
        }
```

The `button6_Click()` function will be invoked when the button “Set Acc” is pressed. The entered value in `textBox2` (Acceleration) will be saved in the variable `eingabe`. `Convert.ToInt32(eingabe)` converts the entered value from string to integer, to calculate with that value which will be saved in variable c.

The calculation describes the converting from pulses to degree (radiant) as it shows in Fig. 11. The variable c will be converted from integer to string, to send the result.

The function `serialPort1.Write()` causes to send via port. The same procedure is required for `textbox4` (Deceleration).

```
private void button7_Click(object sender, EventArgs e)
        {
            string eingabe2 = textBox5.Text;
            int a = Convert.ToInt32(eingabe2);
            a = (int)a * (160 * 360) / 360;

            string tempsp = Convert.ToString(a);
            serialPort1.Write("SP=" + tempsp + "\n\r");

            string eingabe3 = textBox3.Text;
            int b = Convert.ToInt32(eingabe3);
            b = (int)b * (160 * 360) / 360;

            string temppa = Convert.ToString(b);
            serialPort1.Write("PA=" + temppa + "\n\r");
            serialPort1.Write("BG\n\r");
        }
```

The function `button7_Click()` will be invoked when the button “initiate” is pressed and sets the motor in motion. Likewise the function `button6_Click()`, conversions of the values in this part are also carried out and sent via the port with the function `serialPort1.Write()`.

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
        {

            if (comboBox1.SelectedItem.ToString() == "COM 1")
            {
                serialPort1.PortName = "COM1";
            }

            else if (comboBox1.SelectedItem.ToString() == "COM 2")
            {
                serialPort1.PortName = "COM2";
            }

            else if (comboBox1.SelectedItem.ToString() == "COM 3")
            {
                serialPort1.PortName = "COM3";
            }

            else if (comboBox1.SelectedItem.ToString() == "COM 4")
            {
                serialPort1.PortName = "COM4";
            }
        }
```

The function `comboBox1_SelectedIndexChanged()`, is responsible for setting the port. Each port is represented by a “COM” number and is connected to one port. The if-instructions check if the selected item in the combo box is equal to the “COM” number. `serialPort1.Portname` gets the port for communication, including all available COM ports.

6. Tests and results

In this chapter the tests and results of the interface are presented. There have been several tests performed on the interface as the interface control (set the trapezoids, storage and loading the settings as a step or motion), the control of a single motor and the control of ARCHIE.

6.1. Test of Interface Control

In this part, the results of the interface control are presented. The settings of the trapezoid will be taken over by clicking and dragging.

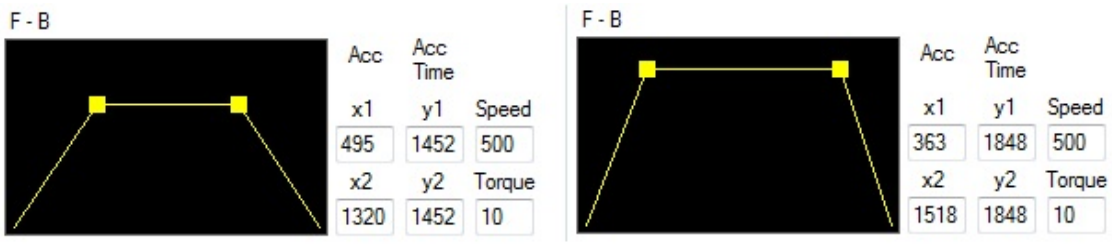

Figure 22: Set values for the trapezoid

The values in the text field will be taken over by the setting of the trapezoid. The trapezoid can also be set by typing in the text boxes if exact parameters are needed (Fig. 22).

Another important function for the test was to store entered values. As it shows in Fig. 23, there is the choice to save as step or motion. The test will show that the storage of the entered data was successful and stored as XML file.

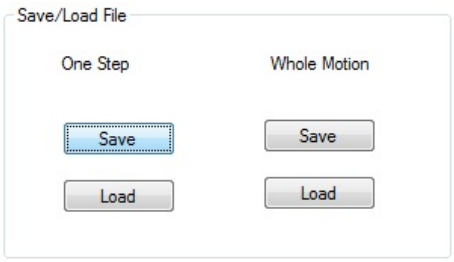

Figure 23: Save/Load File

At least, there will be checked if the values of the entered value are saved as XML file. The XML-files shows the values and the text box name where it is saved. So, if the file is loaded they take over the values and insert it into the appropriate text box by the text box name. Fig. 24 shows the XML-file.

```
<SaveTBDT>
  <ControlName>textBox89</ControlName>
  <ControlValue>1452</ControlValue>
</SaveTBDT>
<SaveTBDT>
  <ControlName>textBox90</ControlName>
  <ControlValue>1452</ControlValue>
</SaveTBDT>
<SaveTBDT>
  <ControlName>textBox91</ControlName>
  <ControlValue>1320</ControlValue>
</SaveTBDT>
<SaveTBDT>
  <ControlName>textBox92</ControlName>
  <ControlValue>495</ControlValue>
</SaveTBDT>
```

Figure 24: XML File with saved values

6.2. Test of Interface on a brushed motor and on ARCHIE

In this part, the controlling of a brushed motor and on ARCHIE is tested. The first goal was to put the brushed motor in movement by using the debug console. The parameters are entered in the console to send the values via a COM port. In this test, the motor is aimed to rotate 45 degrees and back (shown in Fig. 25).

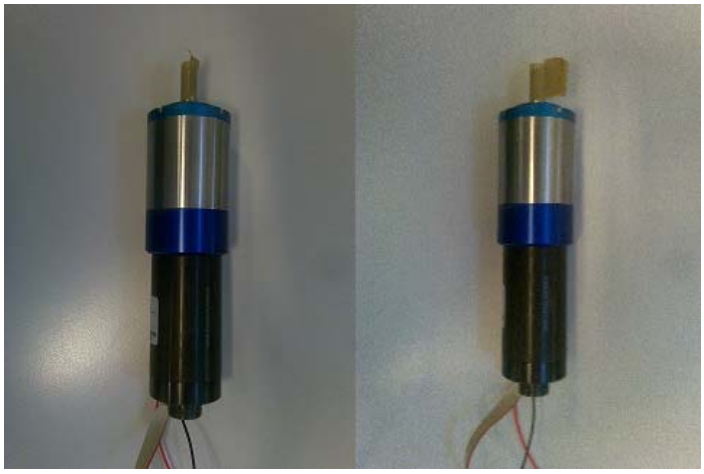

Figure 25: Test on brushed motor

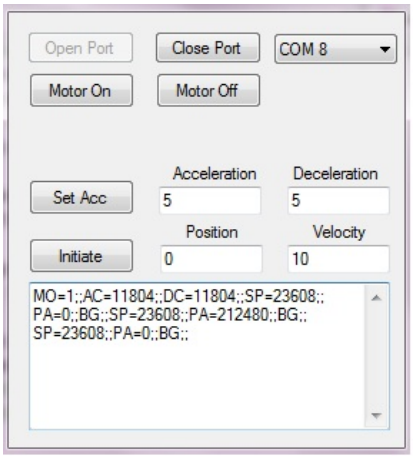

Figure 26: Debug Console

The test on the brushed motor was successful, the parameters were sent and were logged in the textbox. If no values were sent, it would not be logged. The text box notes the entered value and actions (shown in Fig. 26).

The tests have also tried directly on ARCHIE. As with the brushed motor, a motor from ARCHIE is connected to the computer through a serial port. For each motor, one port is needed. Each motor were tested individually and two engines were simultaneously connected to the computer. Again, parameters were entered and sent via the serial port.

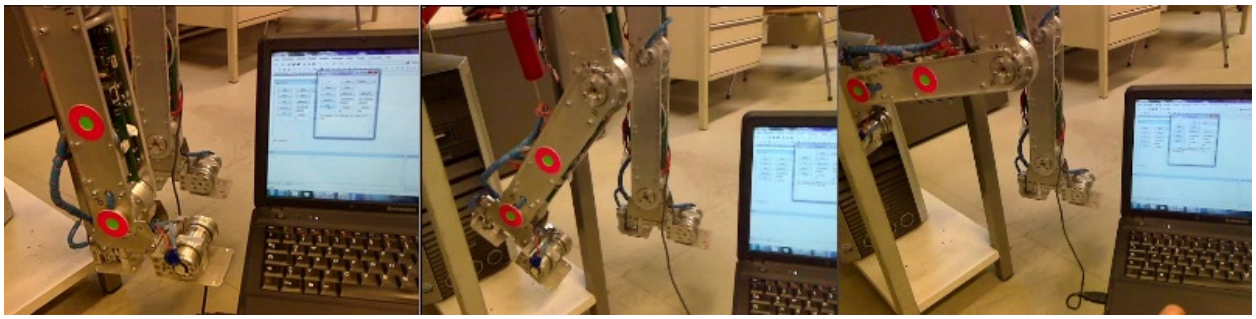

Figure 27: Test on ARCHIE

As shown in Fig. 27, the parameters were successfully sent to ARCHIE. This procedure was carried out for all motors and achieved the same successful results.

7. Summary and Outlook

In this thesis, a flexible and user-friendly interface was designed for a humanoid robot. Some interfaces were presented, that were used for humanoid robots, like the interface of Bioloid, Robonova and Acyut. It has been shown that the essential characteristics must be present in an interface, like the good overview of all the joints, flexible input of parameters and flexible storage of parameters.
It was presented the humanoid robot ARCHIE that was used for testing the interface.
ARCHIE is about 120 cm tall and has 16 engines, which each can be individually controlled.
Furthermore, the requirements of a humanoid robot were presented, for example, the motion controller, the motion planner, motion controller's hardware and the interface requirements in detail. The decision was to use a trapezoidal function, which the velocity, time and acceleration could be set more flexibly.
The interface consists of three levels, in the first level, the trapezoids are set to control the joints, in the second level, the angles of the joints will be set and storage and loading operations can be made. In the third level the set parameters for the joints can be tested or played.
The main work was focused on the implementation of the interface, it was described in detail how the structure of the interface is.
It was necessary to test the application on a debug console. It was a program written for a manual input to check whether the parameters are sent or not. The tests on the debug console showed that the parameters were successfully sent and put ARCHIE in motion.

For level 2, it was planned to program a 3D simulation of ARCHIE, to provide for more flexibility in settings of all parameters. A front view and a side view are available to adjust the joints and to control over mouse. The joints and their initial and final states will be set in the 3D simulation. The parameters are also incorporated in the text boxes. Furthermore, it was also planned to control the humanoid robot in real time over the 3D simulation. As an example, if a leg will be moved in the simulation by mouse, then the robot will be moved at the same time without predetermined parameters.

8. References

Burenhult G. (2000), Die ersten Menschen - Die Ursprünge des Menschen bis 10.000 vor Christus

Byagowi, A. (2010), Control System for a Humanoid Robot, Vienna University of Technology

Erlendur K. (2009), Performance Motion Devices

Tan K. K.., Lee T. H. and Huang S. (2008), Precision motion control: Design and implementation

White R. (2004), So funktionieren Computer - Ein visueller Streifzug durch den Computer & alles, was dazu gehört

Internet Links:

Internet 1 (2010), http://www.techtakeaway.com/articles_robonova1.php (Date: 05.05.2010, 13:00)

Internet 2 (2010), http://acyut.wordpress.com/2008/05/27/gui-graphical-user-interface-upgrade/ , (Date: 05.05.2010, 13:00)