

Freie Universität



Berlin

GRAPHICAL MODELING OF BIPEDAL WALKING WITH AUTOMATIC CODE GENERATION

*Featuring a Simulink Blockset for
Programming of Dynamixel Peripherals*

Manuel Zellhöfer

manuel.zellhoefer@fu-berlin.de

Master's Thesis

Freie Universität Berlin

Fachbereich Mathematik und Informatik

Advisors

Prof. Dr. Raúl Rojas

Dr. Hamid Moballegh

Dr. Tim Landgraf

Berlin, November 17, 2014

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Alle von mir aus anderen Veröffentlichungen übernommenen Passagen sind als solche gekennzeichnet. Die Arbeit ist in gleicher oder ähnlicher Form im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Declaration of Academic Honesty

I hereby declare that the content of this thesis is entirely my own work. I have properly and accurately acknowledged all sources used in the production of this thesis. This work has not been presented, in same or similar form, to any other examination.

Berlin, November 17, 2014

Manuel Zellhöfer

Zusammenfassung

Modellbasierte Entwicklung ist ein prominentes Paradigma im Bereich eingebetteter Systeme. Hierbei bilden Modelle des Zielsystems die Grundlage aus der die weiteren Bestandteile des Systems sukzessive abgeleitet werden. Häufig anzutreffen sind auf diese Weise entwickelte eingebettete Systeme in Steuerungs- und Regelungstechnik-lastigen Anwendungsbereichen wie Industrieautomation, Automobilindustrie, Luft- und Raumfahrt oder (mobiler) Robotik. Eine wichtige Rolle spielen dann grafische Modelle der Dynamik der Systeme und der dazugehörigen Regelungen. Aus diesen Modellen können im Idealfall zugunsten von Effizienz, Fehlerreduzierung und Kostenersparnis automatisiert Quelltexte generiert werden. Diese Arbeit beschäftigt sich mit der grafischen Modellierung und der automatischen Generierung der Steuerungssoftware eines humanoiden Roboters im Zuge des FUB-KIT Projekts. Hierzu wurde die weit verbreitete Softwareumgebung MATLAB mit der eng verbundenen Modellierungs- und Simulationssoftware Simulink derart erweitert, dass die Funktionalität von DYNAMIXEL Aktuatoren mit grafischen Bausteinen in Simulink verwendet werden kann.

Abstract

Modelbased Design is a prominent paradigm in the area of embedded Systems. Models of the target system form the base from which each component of the system is successively derived. Embedded Systems developed this way are often found where control engineering is a vital part like Industrial Automation, Automobile Industry, Aerospace or (mobile) Robotics. An important role have then graphical models of the dynamics and the control of the systems. From those models production code can be automatically generated, ideally with the benefit of improved efficiency, less errors and reduced cost. This work deals with graphical modeling and automatic code generation of the control software of a humanoid robot within the FUB-KIT project. Hereto, the popular MATLAB/Simulink simulation environment was extended as to support the functionality of the DYNAMIXEL actuators using Simulink blocks.

Contents

| | |
|---|-----------|
| 1. Introduction | 1 |
| 1.1. Humanoid Robots and RoboCup | 2 |
| 1.2. Visual or Dataflow Programming, Model-based Design | 2 |
| 1.3. Related Work and Expected Outcome | 3 |
| 1.4. Structure of this Work | 4 |
| 2. Graphical Modeling of Dynamic Systems | 5 |
| 2.1. Central Elements of Signals, Systems and Control Theory | 6 |
| 2.1.1. Systems | 6 |
| 2.1.2. Signals and Processing | 7 |
| 2.1.3. Control | 8 |
| 2.2. From Graphical Modeling to Simulation | 9 |
| 2.2.1. Descriptive Modeling for Design | 10 |
| 2.2.2. Graphical Modeling for Simulation and Analysis in Control En- gineering | 11 |
| 2.2.3. Model Based Design | 12 |
| 2.3. MATLAB/Simulink | 13 |
| 2.3.1. Simulink | 14 |
| 2.3.2. Alternatives | 17 |
| 2.4. Automatic Code Generation | 19 |
| 2.4.1. Application and Benefits | 19 |
| 2.4.2. Realization in Simulink Coder | 20 |
| 3. Bipedal Walking Robots | 23 |
| 3.1. Basics of Robotics and Kinematics | 23 |
| 3.2. Characteristics of Human Gait | 25 |
| 3.2.1. The Gait Cycle and Stability | 25 |

| | | |
|-----------|--|-----------|
| 3.2.2. | Modeling Bipedal Gait | 26 |
| 3.3. | Control Strategies | 27 |
| 3.3.1. | Open Loop Control vs. Closed Loop Control of Humanoids | 29 |
| 3.3.2. | Central Pattern Generators | 29 |
| 4. | Implementation: Starting Point | 31 |
| 4.1. | Configuration of the Robot Platform | 32 |
| 4.2. | Description of the Controller Unit and Actuators | 33 |
| 4.3. | Tools and Workflow in the Project | 34 |
| 5. | Implementation: A Simulink Blockset for Dynamixel Peripherals | 36 |
| 5.1. | Description of the Implemented Blocks | 37 |
| 5.1.1. | Core Blocks | 38 |
| 5.1.2. | Derived Blocks (Subsystems) | 39 |
| 5.2. | Code Generation | 41 |
| 5.3. | Firmware Architecture | 42 |
| 6. | Implementation: Simulink Models to Control the Robot | 43 |
| 6.1. | Open Loop CPG Control | 44 |
| 6.2. | A “Weakly” Closed Loop CPG | 46 |
| 7. | Conclusion & Outlook | 48 |
| A. | Serial Communication with the Microcontroller for Debugging | 51 |
| B. | Fixed Point Numbers and their Application in the Toolbox | 52 |
| C. | Toolbox Setup and Source Structure | 54 |

List of Figures

| | |
|--|----|
| 1.1. Da Vinci's Robot Knight's mechanics | 1 |
| 2.1. System Representation | 6 |
| 2.2. Signal Representation and Filtering | 7 |
| 2.3. Basic Control Block Diagram | 8 |
| 2.4. SysML Internal Block Diagram | 10 |
| 2.5. Block Diagram of PID controller | 12 |
| 2.6. Model Based Software Development | 13 |
| 2.7. Simulink User Interface | 14 |
| 2.8. Simulink Simulation Flow Chart | 16 |
| 2.9. Code Generation, General Process | 19 |
| 2.10. Simulink Coder Code Generation Process | 21 |
| 2.11. PID Controller – Model and Generated Code | 22 |
| 3.1. Kinematic Chain of Two Robotic Legs | 24 |
| 3.2. Phases of the Gait Cycle | 26 |
| 3.3. Inverted Pendulum and the Human Leg's Joints and Muscles | 27 |
| 3.4. Passive Dynamic Walker Principle | 28 |
| 3.5. Physical Central Pattern Generator and its Output | 29 |
| 3.6. Matsuoka CPG Model | 30 |
| 4.1. CAD Rendering of Lower Limb System and Robot | 31 |
| 4.2. DOF of the Lower Limb System of the Robot | 32 |
| 4.3. The CM530 and Structure of the DYNAMIXEL Actuator Network | 33 |
| 4.4. Workflow and Expected Improvement | 35 |
| 5.1. DYNAMIXEL Blockset | 37 |
| 5.2. Read and Write to Dynamixel Blocks and Parameters Window | 38 |
| 5.3. Two Run on Target Block examples and Parameters Window | 39 |

| | | |
|------|--|----|
| 5.4. | Triggered Input Bias Block | 40 |
| 5.5. | Biased Motor Block Model | 40 |
| 5.6. | Code Generation Procedure for the <i>read from DXL</i> Block | 41 |
| 5.7. | Source Code Structure of the Firmware | 42 |
| 5.8. | Main Loop Activity Diagram of the Firmware | 42 |
| 6.1. | CPG Walker Architecture | 43 |
| 6.2. | Open Loop CPG Control | 44 |
| 6.3. | Control Signals for Open Loop Controller | 45 |
| 6.4. | Open Loop CPG Control | 46 |
| 6.5. | Control Signals for “Weakly” Closed Loop Controller | 47 |
| A.1. | Transmit Signal Block Example | 51 |
| B.1. | Fixed Point and Floating Point Binary Patterns | 52 |

List of Tables

| | | |
|------|---|----|
| 2.1. | Common Blocks in Control Engineering Diagrams | 11 |
| 2.2. | List of Modeling Software | 18 |
| 4.1. | Higher Level Communication Methods of the DYNAMIXEL C API | 34 |
| 5.1. | List of Blocks in the DYNAMIXEL Blockset | 37 |

1. Introduction

Robots have been in human's fantasy and imagination for a while. When thinking of science fiction movies, comics or stories, chances are high one will come across some type of robot, interacting with humans just as – or at least: almost as – humans do. The human's wish to create a replica of itself seems strong and old.

Trying to do so dates back to Leonardo da Vinci with his countless inventions and studies done in the late 15th century. One of his contraptions was a robotic knight. A cable operated mechanical apparatus capable of simple motions of arms, head and torso. One can find resemblance comparing da Vinci's invention to blueprints of the mechanical parts of today's humanoid robots. What is different is that by now we are able to control robots, to program them and to make them "intelligent", to whatever extent and whatever that may mean. Of course this is not the only innovation. A lot of work from mechanical, biological, material and computer science contributed in the past century to the development of human-like machines.

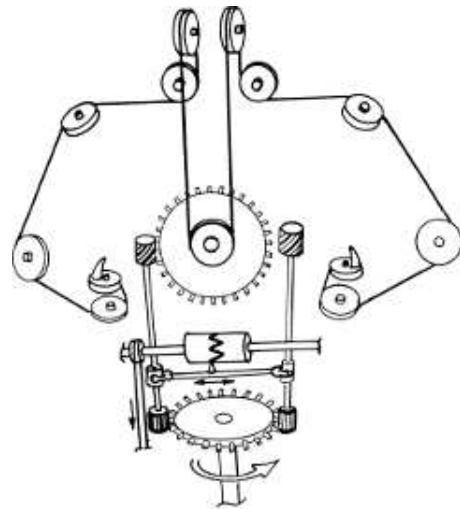


Figure 1.1.: Da Vinci's Robot Knight's mechanics (from [43])

The focus of this work lies on the programming and control of one particular design of a humanoid robot. This single aspect still is comprised by a vast field of topics. The main aim of this work is not more than to provide a tool to facilitate programming, control and the optimization of control algorithms.

1.1. Humanoid Robots and RoboCup

The scope in which this work was done is the so called *RoboCup*. A competition, firstly held in the mid-1990s devised to encourage and push forward research on mobile robotics. International teams of researchers and students gather regularly to test their robots in games of soccer. What previously was chess to the advancement of artificial intelligence may be robot soccer to mobile robotics. As with chess, the long term goal is to play – and ultimately win – against humans. Indeed, the official goal of the organisers is the following:

»By the middle of the 21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup.«

(*The Robocup Federation, <http://robocup.org>*)

Today several leagues exist, allowing for competition in different categories: *Middle Size*, *Small Size*, *Simulation* and *Humanoid*, the latter further branched out into *KidSize*, *TeenSize* and *AdultSize*. The robot used in this thesis belongs to the *Humanoid TeenSize* league. It was developed in collaboration with the Kyushu Institute of Technology in the project hence called *FUB-KIT*.

1.2. Visual or Dataflow Programming, Model-based Design

Now, from the general scope towards the specific topic of this thesis: Graphical modeling of control algorithms. To ease the programming process or – for more complex programs – to even make successful programming possible one often uses diagrams, plans or any other sort of graphical representation. These can visualize the structure and design of a software system. Often, graphical representations of software translate directly to code.

This is where Visual Programming¹ comes into play. Tools that realize this concept don't

¹“the use of visual expressions (such as graphics, drawings or icons) in the process of programming” [5]

offer a text editor as a central workspace but an empty canvas on which the programmer draws diagrams that represent the resulting program. This is said to be an easier approach to programming, enabling it for “non-technical” users. Advanced users benefit from it as well: It allows to quickly devise prototypical algorithms as they are ready to be tested immediately after visually sketching them.

Dataflow programming is a subset of visual programming. As the name suggests, the dataflow of the resulting program is represented graphically. This is what comes to use in this thesis: A control algorithm for a humanoid robot will be modeled graphically. Sensor and actuator signals will be represented graphically as well as systems working with those signals.

In control systems engineering, dataflow Programming has a long tradition in the sense that most well established simulation tools incorporate visual programming in the modeling stage. A very prominent example being MathWorks’ MATLAB and Simulink products. Model-based design aims to raise the level of abstraction of the design process. Systems can be specified more generally and components reused more easily. This helps accelerating design and making it less error prone. [45] This work though does not fully realize model-based design as it does not make use of a simulation. It nevertheless is a step towards that design methodology and its use in the project.

1.3. Related Work and Expected Outcome

For robotics, a few approaches utilizing Visual Programming already exist. Many of them are intended to open robot control for a broader audience (towards entertainment) and to enable educational use of robots in programming. They are often closely linked or part of commercially available robot systems and easily accessible but tied to hardware products. Examples are Microsoft’s Visual Programming Language of the Robotics Developer Studio [8], LEGO Mindstorm’s EV3 and NXT-G tools based on LabView [13] and the Choreographe tool for Aldebaran’s Nao robot [44].

Also, with the exception of LEGO’s tools, the focus lies not within programming and control of the dynamics but rather in programming of static motions (i.e. simple replay of predefined movement patterns) or behavioral programming of the robot. Into these same categories fall the MotionEditor, a key-frame based tool for static motion definition [14]

and the XABSLeDitor, a graphical editor for behavior definition in XABSL² [17].

For some robotic environments, MathWorks offers *Support Packages* for their MATLAB/Simulink programming and simulation environment. These easily allow direct programming of certain hardware from within Simulink by extending it's functionality by respective hardware specific aspects. At the time, these were LEGO's NXT and EV3 platforms as well as the ankle of Aldebaran's NAO robot.

Very similar work for other robotic environments has been done by Kermani for the NAO platform [27] and by Ouchet for ROBOTIS' DARwIn-OP platform [39]. Both developed a MATLAB/Simulink toolbox for usage with the respective robot platform. While being similar in functionality, the overall architectural approach of the control electronics differs too much to be used here.

Eventually, the motivation for this work was to enable a quick and efficient way to program and debug the FUB-KIT robot's control algorithm, the goals of this work finally consisting of the following aspects:

- research graphical modeling of control algorithms and control of humanoid walking
- enable graphical modeling for control algorithms in the FUB-KIT project and potentially similar projects and
- provide an exemplary control algorithm for a humanoid walking robot using a graphical modeling tool.

1.4. Structure of this Work

This work is divided into two parts: The First (up to chapter 3) describes theoretical and methodological backgrounds of graphical modeling of control and control of humanoid robots. It is meant to provide insight for some of the concepts and tools involved and used in the second part. The second part (chapters 4 to 6) describes the software produced in the course of this work. The last chapter is meant to recapitulate the work done as well as to give suggestions for following work.

²XABSL – eXtensible Agent Behavior Specification Language, introduced in [33]

2. Graphical Modeling of Dynamic Systems

»**model** – A simplified or idealized description or conception of a particular system, situation, or process that is put forward as a basis for calculations, predictions, or further investigation.«

(The Oxford English Dictionary)

Graphical representations for communicating and documenting ideas, concepts or structures are elementary in the engineering sciences: Civil engineers draw construction plans, electrical engineers draw circuit diagrams and control engineers draw block diagrams. Graphical modeling in the development of an autonomous humanoid robot can be a valuable tool on three different levels:

1. Descriptive Modeling of the overall system
2. **Modeling of the dynamics of the system or subsystems**
3. Modeling of the autonomous behaviour of the system

Focus of this chapter lies in item 2. Item 1 is briefly touched in section 2.2 to distinguish it from item 2. Item 3 is out of the scope of this work.

This chapter will firstly introduce central elements of signal, system and control theory as basic requirements for modeling of dynamic systems. Coming from there, block diagrams as a prominent mean of describing dynamic systems and their control algorithms are explained. MATLAB and Simulink as a widely used tool for modeling and simulation in engineering is briefly introduced as well as some alternatives and related software for modeling complex systems.

2.1. Central Elements of Signals, Systems and Control Theory

2.1.1. Systems

Systems Theory focuses on modeling, analysis classification and design of systems. A *system* is the abstraction of a process which relates signals to each other. This is a very abstract definition. In accordance with the intuitive meaning, almost anything can be regarded a system: a water tank, the human body, a heating unit or a autonomous robot. [51]

From a technical perspective, essential to the description of a system are

- its **input signals** – describing physical quantities entering the system
e.g. water intake, hormone levels, temperature, sensory input
- its **output signals** – describing physical quantities generated by the system due to the processing of its inputs
e.g. water level, heart rate, heating power, actuator positions
- the **model** of the underlying process
Functional description of the system

Find abstract representations of a system in figure 2.1.

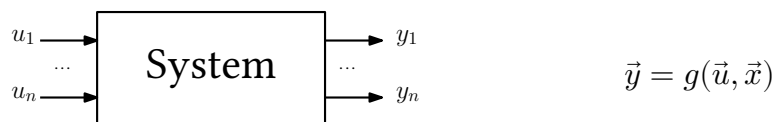


Figure 2.1.: *Left:* Abstract block representation of a system with input and output signals. Complex systems are described by connecting blocks of sub-systems in a block diagram. *Right:* Trivial mathematical notation of a systems relation of input (\vec{u}) and state (\vec{x}) to its output (\vec{y}) via an arbitrary function g .

The functional description depends of the further use and the knowledge about the System. In engineering, mathematical representations in form of differential equations or transfer functions are common. With higher levels of abstraction, block diagrams are

often used. Even a rough informal description of the process might be available when the exact mechanism is yet to be determined¹. [2]

2.1.2. Signals and Processing

A *signal* is the technical representation of a physical quantity. It can be seen as a function with the domain usually representing time and the target set the respective physical quantity. In digital control, signals are represented as a discrete time-series with a certain sample rate and a fixed resolution of the target set (see figure 2.2). If the sample-time is sufficiently small, the fact of the signal being discrete may be neglected. [32]

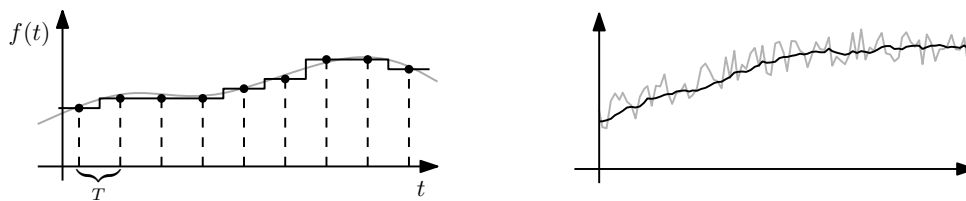


Figure 2.2.: *Left:* Course of an analog, continuous time signal (grey) and a possible digital representation after sampling with sample-time T (- - -).

Right: An example for a noisy signal before and after filtering with a moving average. filter

Sensors transform physical quantities into signals so they can be handled by a digital control system. A sensor usually has an electrical element that generates a voltage depending on the measured quantity. In order to provide a digital signal, the voltage is then converted into a series of digital values by an analog-digital converter (ADC). In some cases digital values are directly at disposal (e.g. switches or certain angle sensors). In robotics often used sensors are force sensors (e.g. to measure load or weight distribution), accelerometers or magnetometers (used to measure orientation or position) or angle sensors (to measure the angular position of a joint).

Signal generators provide well-defined waveforms that, while mostly used for testing or (wireless) communication, may be necessary or at least useful in more complex control-algorithms when those need periodic signals. Examples are sine, sawtooth, triangle or pulse waves. The signals are mostly defined by amplitude and frequency.

¹The procedure of technically determining the underlying mechanism is called System Identification.

When coming directly from the sensor, the signal is always erroneous to a certain degree. Bias, drift, loss or noise may render the signal difficult to use for control. Sensor systems may account for that before or after digitization with the use of *filters*. It may still be necessary to further process or transform a signal to use it in a control system. Filters are able to account for some types of noise (see figure 2.2). They range from simple ones which only look at a few samples of the time-series using simple relations for determining the output signal (e.g. low-pass and median filter) to more sophisticated filters involving more complex models (e.g. Kalman-filter and adaptive filters). [49]

2.1.3. Control

Control is the manipulation of the input signals of a given system in order to achieve a desired output. One can distinguish between two main strategies: Open loop or closed loop control. For closed loop control, the output of the system is used by the controller to determine the new input (i.e. feedback). For open loop control this is not the case, the input from the controller is based on a quantity foreign to the system. For this to work, sufficient knowledge and predictability of the system to be controlled is necessary. [2]

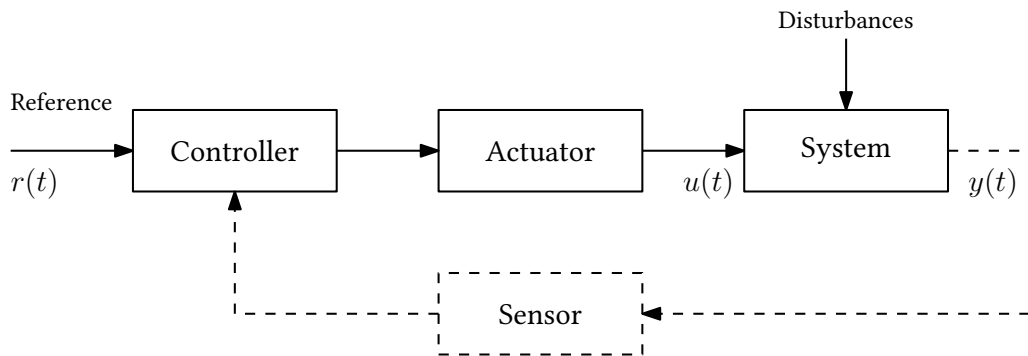


Figure 2.3.: Basic block diagram for open loop (-) and closed loop (- + - -) control.

A most simple and very common controller is the so-called proportional controller. Its output is formed by multiplying the *error* e of the system (the desired reference value r minus the current output y of the system) by a *proportional factor* K_p :

$$u(t) = K_p \cdot e(t) = K_p \cdot (r(t) - y(t))$$

To account for some drawbacks (oscillations, steady-state error) of this simple controller,

one can bring into account both integral and derivative over time of the error e :

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t)$$

The integral term eliminates the steady-state error, the derivative term can improve stability. This kind is called *PID controller* for the proportional, integral and derivative contributions to its output value. [32]

The elements that translate signals back into a physical quantities are called *actuators*. Based on its input from the controller an actuator manipulates the system accordingly. Typical actuators are valves (for controlling flows) or motors (to manipulate positions).

Controllers can vary largely in complexity while still being governed by the same basic principles. We may for example find a PID controller in a DC motor of the wheels of a mobile robot while the higher level control algorithm for positioning the entire robot may be implemented – again – as a PID-type controller. This is one example of reusability found in the design of complex systems which can be more easily exploited by the use of model-based design in conjunction with code-generation (see section 2.4.1).

2.2. From Graphical Modeling to Simulation

The systematical usage of block diagrams for modeling in engineering rose alongside with the increasing complexity of technical systems in the second half of the 20th century. By the same time the discipline system engineering became increasingly popular. Engineers had not only to deal with problems from either domain but also account for the integration of mechanical, electrical and later hard and software engineering. [4] Further momentum and a broader audience gained this trend for integration with the advancing miniaturization of electronic devices and the accompanied *embedding* of software within the connected, “tangible” systems. Any autonomous robot system essentially is an embedded system. Another important aspect of engineering is simulation, which is closely linked to modeling: The created models are evaluated and analyzed which allows for early and cheap validation and optimization.

2.2.1. Descriptive Modeling for Design

Regarding graphical modeling in a software context, the Unified Modeling Language (UML) comes to mind. UML is a very extensive, standardized language for modeling of complex systems. Initially designed for software systems, its generality allows for modeling in a wide range of domains. The diagrams specified in UML belong to two categories:

- *Structural diagrams* describing the static, architectural properties of a system and
- *Behavioral diagrams* describing the dynamic properties of a system

Another language based largely on UML and extended to specifically allow modeling of complex hybrid (software and hardware) systems is SysML. For an example diagram displaying its capability to model a mechanical system see figure 2.4. [48]

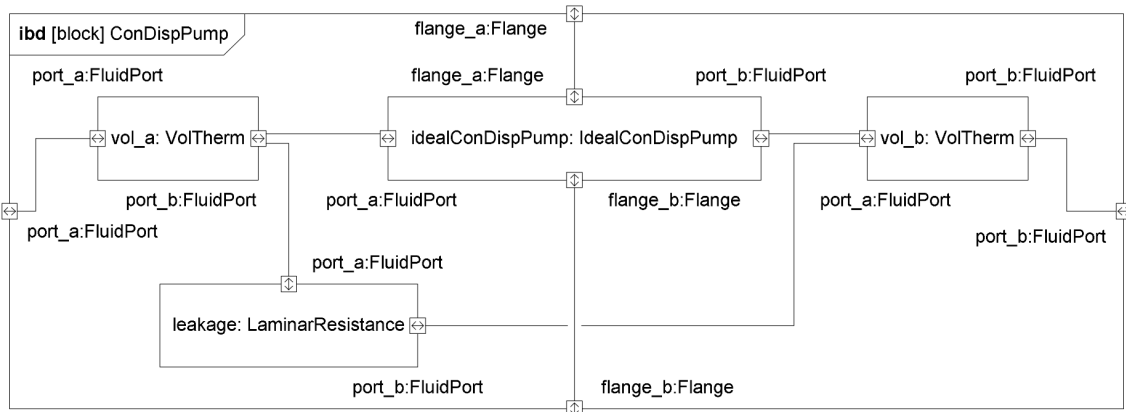


Figure 2.4.: An example of a SysML internal block diagram. The diagram models a pump, involving submodels of an ideal pump as well as resistance and thermal terms. (figure from [25])

The primary goal of modeling a system via UML or SysML is to cope with the large complexity of the to-be-designed system. Modeling in this regard helps to identify affordances, dependencies and ultimately procedures on the way to implementation and synthesis of the system.

2.2.2. Graphical Modeling for Simulation and Analysis in Control Engineering

While SysML and UML are widely used for descriptive modeling of systems, they are not well suited for modeling continuous dynamic systems in order to simulate their real-world physical behavior. However, approaches to integrate or combine both aspects exist. [1, 25] Simulation environments mostly incorporate their own modeling language (see section 2.3) which –when graphical– is largely based on block diagrams used in the respective domains.

A block diagram as used in control engineering essentially consists of

- nodes called blocks, representing subsystems or elementary mathematical operations and
- edges representing signal flow.

Table 2.1 shows some elementary blocks and their corresponding mathematical expression. Subsystems are usually a group of elementary mathematical operations modeling a distinguished aspect of the whole system thus increasing clearness and expressiveness of the diagrams.

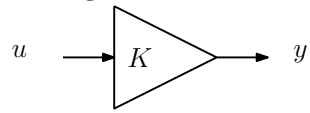
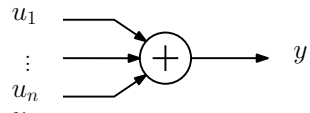
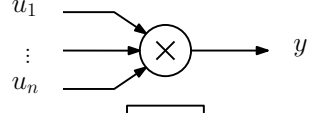
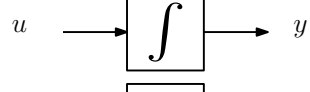
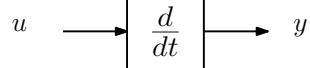
| Pictogram | Description | Mathematical Function |
|---|-------------|----------------------------------|
|  | Gain | $y = K \cdot u$ |
|  | Sum | $y = \sum_i u_i$ |
|  | Product | $y = \prod_i u_i$ |
|  | Integral | $y = \int_{t_0}^t u(\tau) d\tau$ |
|  | Derivative | $y = \frac{d}{dt} u(t)$ |

Table 2.1.: Common Blocks in Control Engineering Diagrams

Both, block diagrams and a set of differential equations can equivalently describe the

dynamics of a system. As an example see figure 2.5, showing a block diagram describing the differential equation from section 2.1.3. [2]

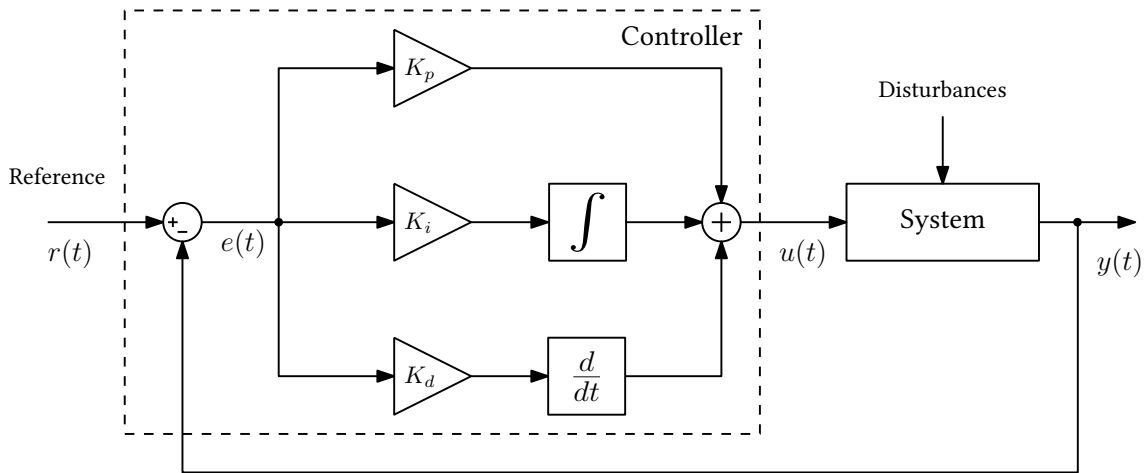


Figure 2.5.: The PID-Controller from section 2.1.3 within the basic control loop according to figure 2.3

Simulation of the model is done by solving the differential equations implied by the layed out model numerically. Tools often provide a variety of solvers for differential equations allowing for adaption to the respective goal of the simulation. Input signals to the systems are similarly chosen or generated to enable the desired goal of the simulation.

2.2.3. Model Based Design

The comparatively new paradigm of model based design tends to speed up the development and design process. This is done by not only using a model to derive specification or ease design but by integrating the system model closely into every stage of the development process. Modeling and Simulation of the dynamics of a hybrid system thus become more involved in the overall process (see figure 2.6). Model based design is motivated by the fact, that all subsequent development artifacts and products following the specification are derived by iteratively refining the initial model. Its ultimate goal being to automatize this refinement process. [38, 46]

Claimed advantages of model based design (based on [45]) are

- easy redeployment after design changes, rapid prototyping

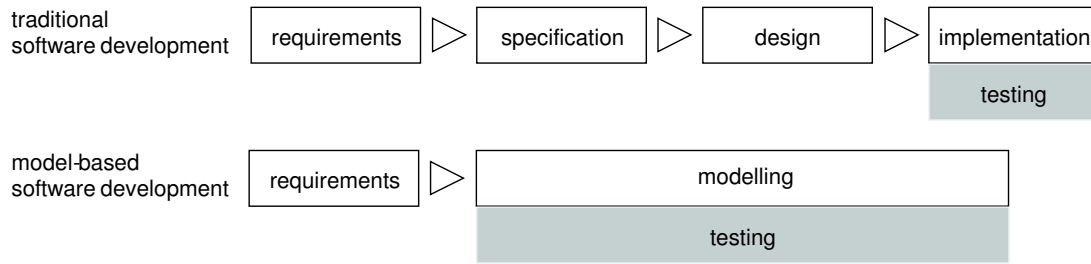


Figure 2.6.: Traditional software development compared to model based development. Modeling and simulation may occur in stages of traditional development however not as central as in model based approaches. (figure from [7])

- early and easy verification and analysis
- focus on problem solving in the systems domain rather than on programming
- raising abstraction

2.3. MATLAB/Simulink

MATLAB is a commercial, extensive software environment for scientific and technical computing, visualization and programming. Its core functionality consists of easily accessible, optimized and specialized algorithms from linear algebra, basic statistics, numerical analysis and filtering. It is extended by toolboxes which provide additional functionality in a variety of fields (symbolic math, signal processing, control systems, etc.).

MATLAB is a high-level interpreted programming language. Its semantics is based on linear algebra, matrices and vectors being the main datatypes. MATLAB is often used to sketch, prototype and evaluate algorithms that involve the provided mathematical environment before they are implemented in a low-level language like C. This especially applies to digital signal processing, or control algorithms as they usually are developed to run on microprocessors or other dedicated hardware. Solutions to facilitate this process are provided by the developer of MATLAB: Toolboxes that provide code generators, special datatypes or tools to accelerate the workflow. [21]

2.3.1. Simulink

Simulink is a modeling and simulation environment integrated into MATLAB. It supports graphical design and modeling of systems via hierarchical block diagrams and simulation of the designed models. A library of blocks provides basic and more specialized functions and subsystems. It is extendable via blocksets analogous to the toolboxes for MATLAB.

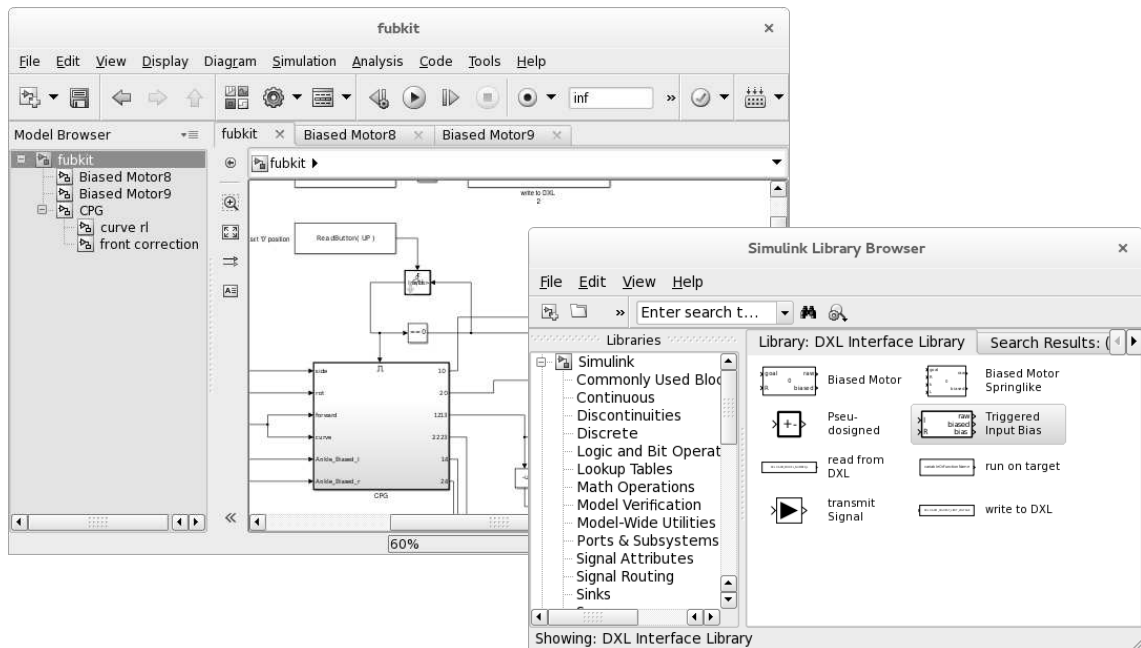


Figure 2.7.: A screenshot of the Simulink user interface with the main tools: the canvas for drawing the block diagram and the block library browser.

Simulink provides a data flow oriented programming language. The creator of MATLAB and Simulink is putting a significant effort into promoting model based design and Simulink as a tool tailored to use this paradigm. This section is supposed to give some insight into how Simulink executes models with regard to the creation of custom blocks and how they interplay with the Simulink engine. The only source is the products documentation.

Simulink Blocks

A block in the Simulink language is defined similarly to the definition of a system in section 2.1.1: a set of inputs, states and outputs. Further routines to

- determine outputs,
- determine derivatives and
- update the states

based on input and state of the block and current simulation time. Additionally, there are routines that are needed to comply with the simulation engine for providing e.g. sample time, value dimension and datatype or system parameters. See figure 2.8 for a complete sketch of the initialization and simulation process.

For the creation of custom blocks, Simulink provides different methods varying in flexibility and complexity:

- *via MATLAB-function*

Simulink offers a block that allows for textually describing a system using a MATLAB function. Inputs and outputs of the system are arguments and return values of the MATLAB function. This is the least flexible method.

- *via MATLAB S-function*

Using a structured MATLAB function, Simulink exposes access to a larger part of its simulation environment. Two differently abstract interfaces (“Level-1” and “Level-2”) exist.

- *via C/Fortran S-function*

Finally, Simulink allows for definition of S-function in lower level languages C and Fortran. All callbacks for a system definable in Simulink (see figure 2.8) are accessible.

Simulation

At the beginning of the simulation Simulink determines the order of the blocks calculations based on dependencies among them. The user can assign priorities, which are evaluated where ambivalences exist. During simulation, the outputs and eventually states of

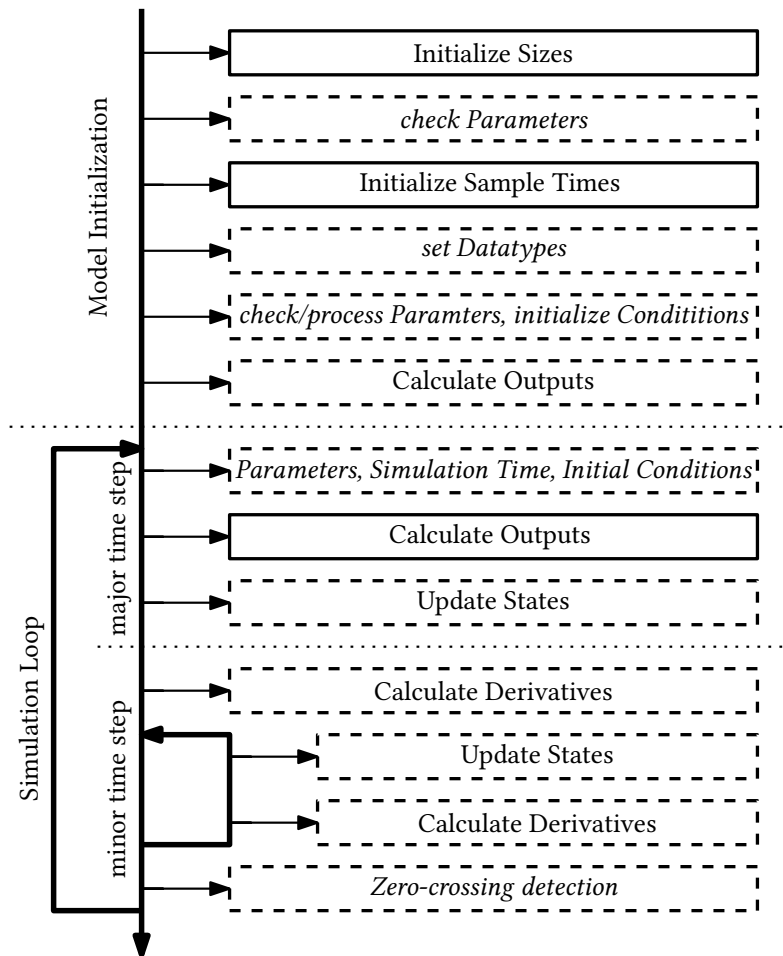


Figure 2.8.: Flow chart describing Simulink model initialization and simulation loop procedure from the view of a blocks implementation. Dashed callbacks and elements are optional. Italic text represents a group of callback methods. Simulink divides the simulation loop in a major and a minor timestep: Continuous states are integrated in the minor timesteps to reach desired accuracy. (based on figure in [24])

the models blocks are calculated for successive time steps. When Simulink blocks have states, they can be classified by their temporal representation:

- *Discrete* for blocks that change state at fixed time steps (e.g. Unit Delay) and
- *Continuous* for blocks that change state continuously (e.g. Integral, Derivative)

Discrete blocks are updated at the next time step using the corresponding sample time and difference equations. For continuous blocks, the timecourse of the states is calcu-

lated using solvers for ordinary differential equations (ODE-solvers). Simulink offers a variety of such solvers, each with their strengths and weaknesses with respect to structure and time constants of the ODEs resulting from the block diagram. [22, 23]

2.3.2. Alternatives

Luckily, MathWorks sponsored the 2014 RoboCup by providing participating teams with licenses to their software. Nevertheless a brief look outside the box seems reasonable. Mostly, for not appearing like a fervent MathWorks advocate but also to clarify the relevance of model based design outside the preprocessing sphere of the MathWorks products ([29]).

Several other software environments exist for modeling and simulation of dynamic systems, most of which being commercial software. *Scilab* and its modeling and simulation environment *Xcos* being the free software found closest to resemble MATLAB/Simulink in functionality. However, Scilab/Xcos lacks in provided blocks, comparatively poor documentation and embedded code-generation capabilities that only exist as proof of concept.² Commercial forks that put effort in advancing this capability exist (Evidence's E4Coder³ and Equalis' Coder⁴).

Direct competitors to MATLAB/Simulink in any regard are e.g. Dassault Systemes' Dymola, National Instruments' MATRIXx, Wolfram's SystemModeler or Maplesoft's Maplesim. Other solutions are more specialized towards a certain domain (ASCET, automotive) or integrate descriptive and dynamic modeling (SCADE) presumably being more complex. Two notable open source projects are NASA's relatively new openMDAO framework and UC Berkeley's relatively mature Ptolemy project being in development since the early 1990's. Table 2.2 shows an overview with no claim for completeness.

²The Gene-Auto project (<http://gene-auto.org>) implemented embedded code generation for Simulink and, albeit with less priority, Xcos. Commercial distributions of Scilab/Xcos build upon that.

³<http://www.e4coder.com>

⁴<http://www.equalis.com>

| Software | Custom Blocks | Generated Code | Embedded Support | Toolchain Support | First Version | Latest Version | Open Source |
|--|----------------------|-----------------------|-------------------------|--------------------------|----------------------|-----------------------|--------------------|
| SystemModeler http://www.wolfram.com/system-modeler | Modelica | ✗ | | | 2011 | 4.0 (2014) | |
| MapleSim http://www.maplesoft.com/products/maplesim | Maple, Modelica | C | | | 2008 | 6.4 (2014) | |
| MATRIXx http://ni.com | BlockScript | C | ✓ | | ~1994 | 8.1.7 (2012) | |
| Dymola http://www.3ds.com/ | Modelica | C | | | 2003 | 2015 (2014) | |
| MATLAB/Simulink http://mathworks.com/simulink | MATLAB, C, Fortran | C/C++ | ✓ | ✓ | 1990 | 2014a | |
| Scilab/XCos http://scilab.org | Scilab, C, Fortran | C | | | 1994 | 5.5 (2014) | ✓ |
| openMDAO http://openmdao.org/ | Python | ✗ | | | 2010 | v0.10.1 (2014) | ✓ |
| Ptolemy II http://ptolemy.eecs.berkeley.edu/ptolemyII | Java | C | | | 1996 | 10.0SVN (2014) | ✓ |
| MLDesigner http://www.mldesigner.com/ | - | C | | | 2000 | 3.0.0 (2014) | |
| SCADE http://www.esterel-technologies.com/products/scade-suite | o | C, Ada | | | | | |
| ASCET http://www.etas.com/de/products/ascet_software_products.php | o | C | | | 1997 | 6.2.0 (2013) | |

Table 2.2.: List of actively developed modeling software.

Custom Blocks – How can custom blocks be integrated

Generated Code – Language(s) available for code generation

Embedded Support – Specific support for embedded platforms available

Toolchain Support – Allows direct integration of custom toolchains for compiling of generated code

The dates were gathered from the developers websites.

2.4. Automatic Code Generation

This section gives an overview over the basic process of automatic code generation based on models of dynamic systems and its applications integrated in simulation environments. Most environments as discussed in the previous section integrate automatic code generation. For Simulink models, there are at least two code generator tools developed aside MathWork's own product (Simulink Coder⁵): dSPACE⁶ TargetLink and the open source project Gene-Auto⁷. An early example for code generation from block diagrams can be found in [41] where the predecessor of Ptolemy II from table 2.2 was used and code for a digital signal processor generated.

A code generator is comparable to a regular compiler that transforms a higher level language into machine code. However the graphical model as input for the compiler is far more complex than textual source code. [46] All mentioned code generators follow a basic principle: The graphical model is parsed and converted into an intermediate representation. From this intermediate representation the hardware dependent code is generated. [46, 47, 18]

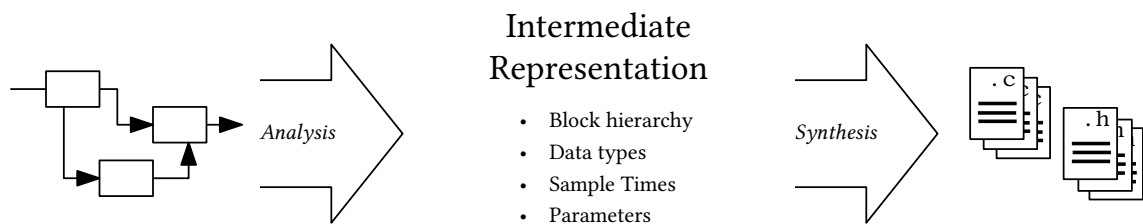


Figure 2.9.: Code generation, general process. Analysis and Synthesis phase are present like in compilers for higher level languages.

2.4.1. Application and Benefits

Regarding modeling and simulation environments, code generation finds two general application scenarios⁸:

⁵Simulink Coder was earlier named Simulink Real Time Workshop (RTW)

⁶<http://www.dspace.com>

⁷<http://geneauto.org>

⁸Another application for code generation exists in conjunction with descriptive modeling: Tools generate template or production code for applications modeled in e.g. UML.

- speed up the simulation on the host⁹ computer and
- create production code from the modeled algorithms for use outside the simulation environment.

Simulation generally occurs interpreted inside the simulation environment. Running the simulation using a native executable can significantly improve the speed. For this, code generators in such environments often allow for generating host-specific low-level code and even executables that may run independently of the simulation environment. The speedup accelerates techniques like automated parameter tuning and optimization of the control algorithms and makes approaches involving computationally intensive techniques like machine learning and genetic programming feasible. More common is the use of such sped up simulation for testing of the deployed or prototyped device using the host as replacement for the to be controlled system with the involving real time constraints¹⁰ (i.e. Hardware-in-the-Loop testing).

Relevant to this work is the creation of production code: The difference to the previous scenario is the absence of host-specific elements and calls to eventual runtime environments in the generated code and in contrast the ability to include target-specific code and optimizations. As in table 2.2 some environments even include toolchains and cross compilers enabling the build process from the model down to the firmware binary. This accelerates the whole development process of control algorithms substantially and completes the Model Based Design principle, eliminating the need for “manually” programming the once modeled control algorithm. A further notable advantage is the presumed correctness of the generated code. The responsibility for the correctness of the code shifts at least partially from the designer or implementor of the control algorithm towards the implementor of the code generator keeping their each necessary specialization narrow.

2.4.2. Realization in Simulink Coder

This section merely summarizes the relevant parts of the documentation of the Simulink Coder product in order to point out the steps necessary to integrate new blocks supporting code generation. The Analysis phase of the code generation process closely

⁹In embedded programming the platform for development is usually called *host*. The processing unit that will be running the developed algorithm is called *target*.

¹⁰Hence, presumably, the early naming of the Simulink Coder product (“Real Time Workshop”)

resembles the simulation procedure as in figure 2.8. Instead of simulating the system, an intermediate representation is generated. The simulation loop is replaced by a single call to a separate method of the block’s implementation (“*writeRTW*”) that allows to specify additional parameters to be written to the intermediate representation. [24]

The intermediate representation is a text file (named *model.rtw*) containing an hierarchical data structure composed of key-value pairs. In the synthesis step it is used as input to the *Target Language Compiler*. This compiler modifies the intermediate representation and generates output code files. It does so by using **.tlc* files that describe transformation and output rules for

- the desired target-system (this **.tlc* file is a parameter to the compiler and acts as an entry point) and
- each available atomic block.

Figure 2.10 shows the entire process. [19]

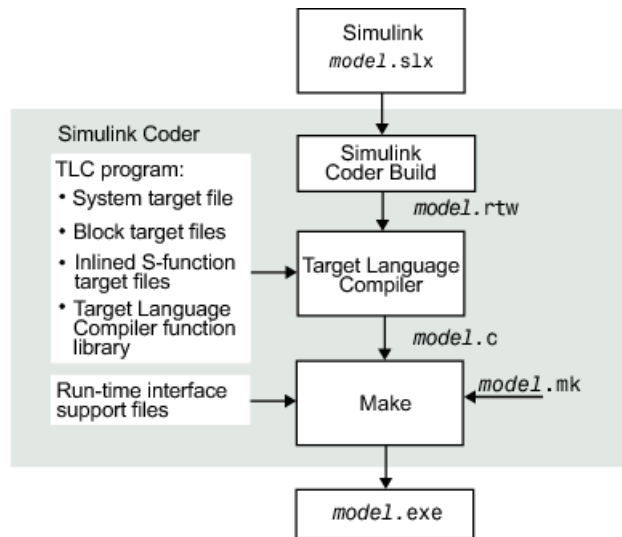


Figure 2.10.: Complete code generation process in Simulink Coder, from model to a Windows OS executable. (figure from [20])

Therefore, in order to add support for a specific platform to Simulink Coder one might create a corresponding system-**.tlc* file. Generic and common target configurations are shipped with the product. For this work, in order to add support for a custom block, block-**.tlc* files are necessary (as well as custom blocks implementations as in sec-

tion 2.3.1). Figure 2.11 shows some example code generated from the PID controller.

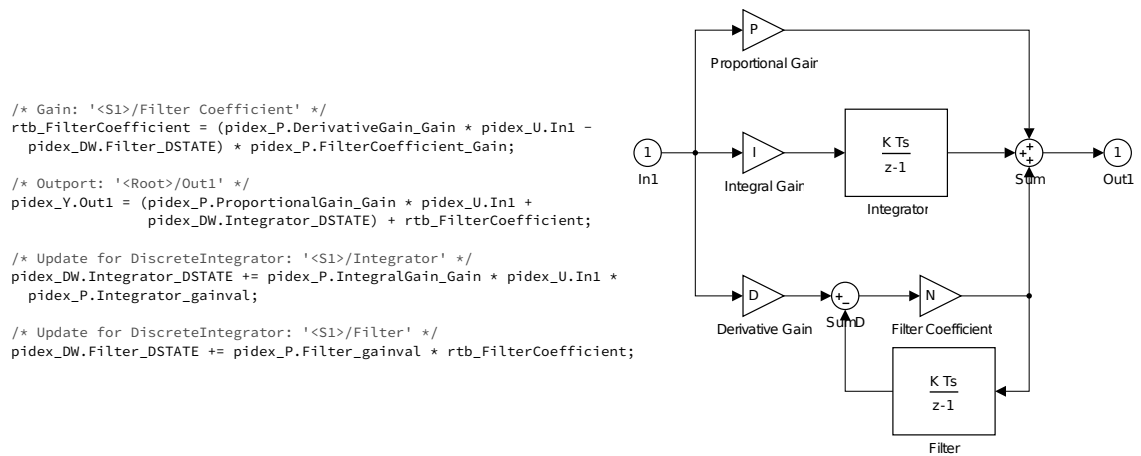


Figure 2.11.: Simulink library’s block implementation for a discrete PID controller alongside with the generated code by Simulink Coder (relevant lines inside the model’s step function only).

3. Bipedal Walking Robots

The development of a robot that is able to get around like a human on two feet is one of the big challenges in mobile robotics. The stability and security humans have on two feet while being able to move quickly and on changing terrain is still considerably out of reach for humanoid robots. This chapter is supposed to summarize the peculiarities bipedal locomotion entails, outline the basic approaches developed to date and give a look at a central structural element the implementation in this work is built around: Central Pattern Generators (*CPGs*).

3.1. Basics of Robotics and Kinematics

Kinematics is the mathematical basis to model the movements of robots. It is the study of motion of bodies while neglecting the forces that cause the motions. Thus the only quantities being considered are angular and translational positions, speeds and acceleration.

Kinematics is essential to relate the geometrical pose of an articulated robot to the configuration of each existing joint. Two problem statements are central:

1. *Direct Kinematics:*

$$(x, y, z, \alpha, \beta, \gamma)^T = f(\vec{\theta})$$

Given the angles of all joints, where in space will a point of the robot be?

2. *Inverse Kinematics:*

$$\vec{\theta} = f^{-1}((x, y, z, \alpha, \beta, \gamma)^T)$$

Given a point in space what are the joint angles necessary to reach it?

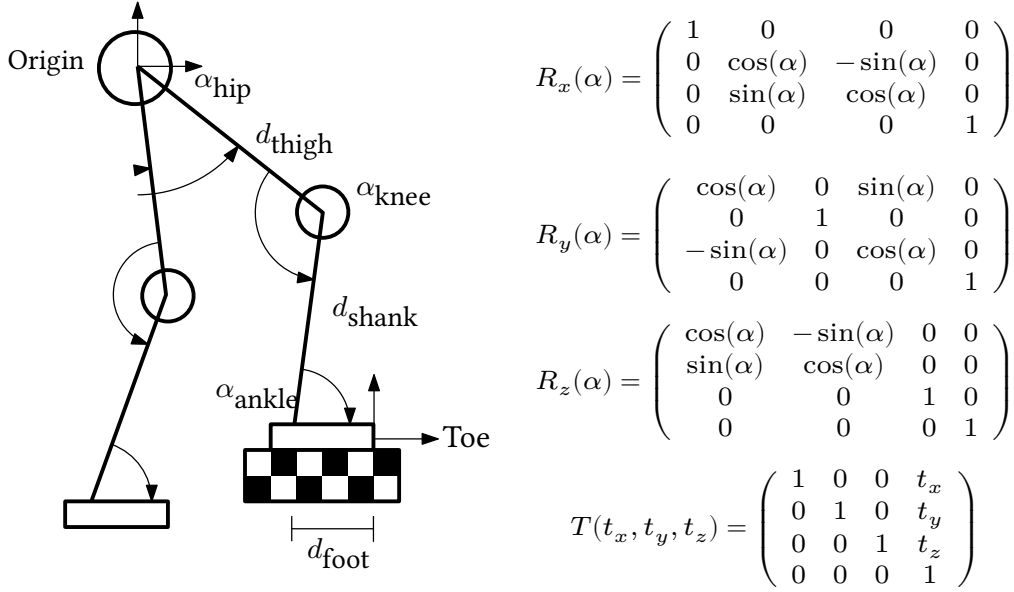


Figure 3.1.: *Left:* Kinematic chain of two robotic legs. The position of the foot on the target is found by following the rotational (joints, α) and translational (limbs, d) elements starting from the origin.
Right: Transformation matrices for 3 dimensions in homogenous coordinates, $R_x(\alpha)$ being the rotation matrix for rotation by α around axis x .

Finding solutions for direct kinematics is a straight forward task. Given a origin coinciding with a suitable point on the robot and all set joint angles the position of any point on the robot in space is found by following the kinematic chain from origin to the desired position on the robot. Using homogenous coordinates, this task consists of a series of matrix multiplications. Considering schematically the “robot” in figure 3.1, R_z being the rotation matrix around axis z and T_x translational matrix along the natural direction of the corresponding link

$$T_O^T = R_z(\alpha_h) \cdot T_x(d_t) \cdot R_z(\alpha_k) \cdot T_x(d_s) \cdot R_z(\alpha_a) \cdot T_x(d_f)$$

describes the transformation from the “Origin” frame to the “Toe” frame. Multiplication of a vector representing a point in the “Toe” frame with T_O^T results in a vector to the same point inside the “Origin” frame. This provides the cartesian coordinates x, y, z , the orientation α, β, γ is easily derived. A formalism to derive transformation matrices systematically and unambiguously is provided by the Denavit-Hartenberg convention.

For the inverse kinematics problem, there is no generally applicable solution. To every given link-joint configuration of an actuated robot a specific solution needs to be developed. There are three approaches to do so:

- *Algebraic*
By inverting the forward kinematics equations, which is difficult or even yields unusable solutions
- *Geometric*
By exploiting the geometric configuration of the kinematic chain using trigonometric laws
- *Iterative*
By using a linearization of the forward kinematics equation and different optimization methods

Determining the inverse kinematics has to be performed quickly, especially for controlling humanoids. Iterative approaches, are usually computationally expensive. Most common are probably geometric solutions as done by Johannes Kulick [28] for a recent version of the kid size humanoids at FU Berlin.

An approach useful for humanoid robotics which this work highly benefits from, is to use a configuration of joints, links and actuators that facilitates the intended motions of the limbs (i.e. stepping or swinging the leg for humanoids). Section 4.1 describes an example.

3.2. Characteristics of Human Gait

3.2.1. The Gait Cycle and Stability

Bipedal locomotion consists of a repeating sequence of motions performed by the limbs. The repeating pattern can be divided into phases. The human gait cycle can be divided into four different phases as depicted in figure 3.2:

- *Double support phase* – Both feet on the ground,
- *Swing phase* – One foot on the ground, the other swinging

- intermediate phases before and after swing.

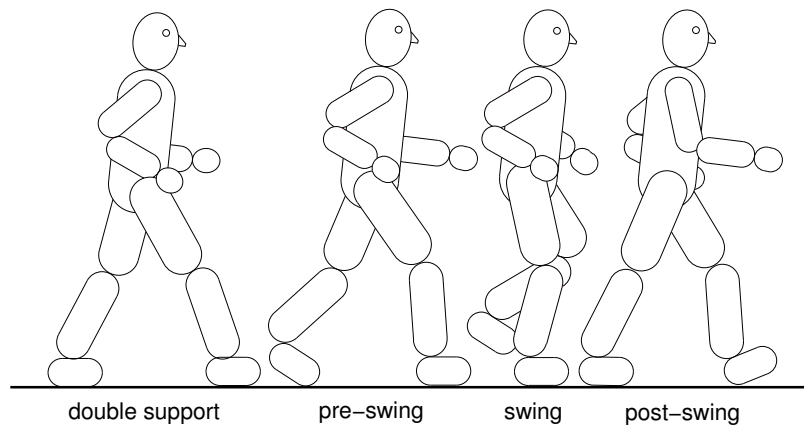


Figure 3.2.: Four phases of the gait cycle. (from [49])

Several approaches exist, to mathematically formalize stability of postures during gait¹. These are derived from the position of the center of mass, the center of pressure and the forces acting on the ground. Regions in which these quantities allow for stable posture are called *supporting area*. Walking techniques can further be characterized as *statically balanced* or *dynamically balanced* referring to these quantities residing in the supported area all the time (*statically balanced*) or temporarily leaving this area (*dynamically balanced*).

Statically balanced gait results in very slow, unnatural locomotion (resembling sneaking in some way) and requires a considerable effort in both motion planning and joint force exertion. [34] Dynamically balanced gait allows for faster, more energy efficient locomotion.

3.2.2. Modeling Bipedal Gait

A common and most simple model employed to describe the dynamically stable gait is the inverted pendulum model (see figure 3.2): The robot is treated as a stiff pendulum. The center of mass of the robot coincides with the weight and the support foot coincides with the pivot point of the (virtual) pendulum. The double support phase is negligibly short. This model is used to derive kinematics for very simple possi-

¹for a short summary see [49, p. 9ff] or [36] for a more comprehensive treatment

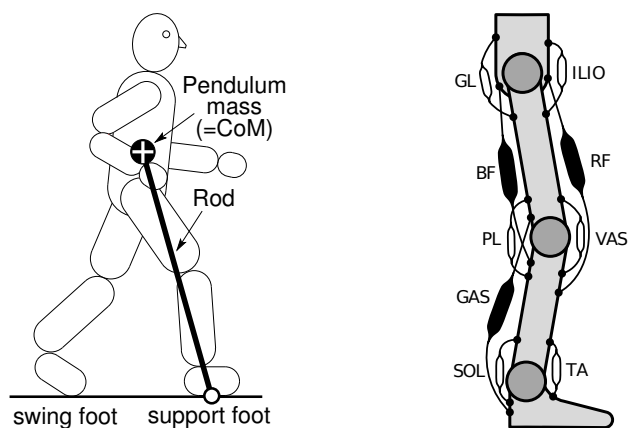


Figure 3.3.: *Left:* Inverted pendulum model for gait.(from [49])

Right: Leg joints and most relevant muscle groups. Monoarticular – *white* ,
biarticular – *black* (based on figure in [42])

bly even non-actuated² walkers and analyse energy distribution and exchange during gait. [26, 11, 12, 34]

A more sophisticated model takes the real structure of the human leg into account and incorporates segmentation and the role of muscles and tendons. The human leg consists of three segments: *thigh*, *shank* and *foot*. The segments are linked by two joints: *knee* and *ankle* and the whole leg is linked to the torso at the *hip*. For any of the three joints two muscle groups exist that exert force in opposite direction to enable movement about the respective joint (“antagonist-agonist configuration”). Further, muscles exist that span over more than one joint: Biarticular muscles which play a important role for energy transfer during gait and synchronization of the connected limbs. This unique configuration is known to effectively support stabilization and energy effectiveness. For a more exhaustive treatment see [42].

3.3. Control Strategies

Losely based on [49], control strategies can be categorized into three types, with the main distinction being the underlying gait model:

- Control strategies predominantly relying on an accurate model to (ideally) ensure

²Walking downhill with only gravity as energy source.

stability at all time. Usually closed loop control architectures where significant effort has to be put into correctly estimating the current state of the robot and fully control every joint of the robot. Examples are *Zero Moment Point* based control algorithms.

- Control strategies building upon simpler models relying on an overall stability of a periodic stepping pattern and exploiting the inherent instability of gait. Locomotion is realized by carefully deviating from a stable cycle by selectively actuating selected joints. Examples are based on the *Passive Dynamic Walker*.
- Control strategies that are “nature inspired” involving neural nets or machine learning strategies. These approaches rely less on concrete models of the locomotion process and gear more towards exploiting biomechanical and physiological structures found in the human body. Approaches based on *Central Pattern Generators* are a common example.

This categorization however is not completely distinct. Often principles from one category are applied on algorithms of the other in order to improve certain aspects.

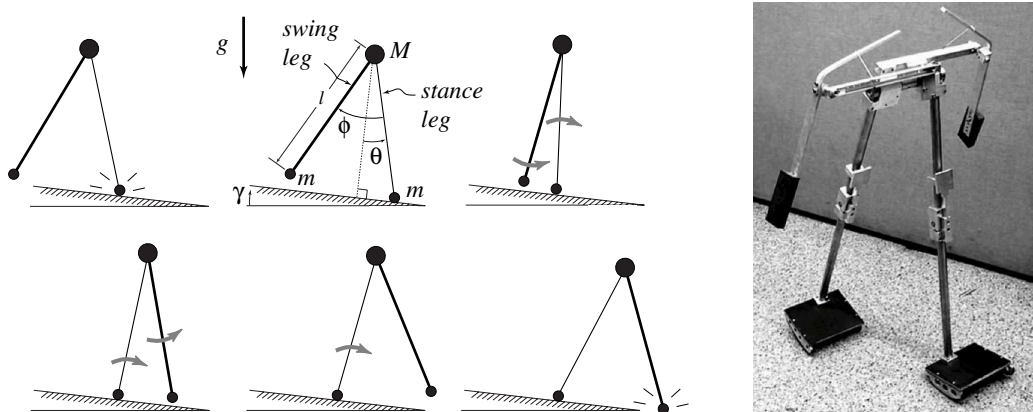


Figure 3.4.: Step for an unactuated simple passive dynamic walker, stably walking downhill with only the gravitational force as source of energy. This mechanism actuated results in a stable walker. Applicable control strategies are either based on simple models or based on central pattern generators. (Left figure from [11] and right from [6])

3.3.1. Open Loop Control vs. Closed Loop Control of Humanoids

Open loop and closed loop control are two fundamentally different control strategies (see section 2.1.3). Locomotion of humans clearly is a closed loop controlled process with it relying on a multitude of sensory feedback: Ground contact and impact of the feet, input from the vestibular system and posture of the limbs. For the technical realization of gait however both strategies, open and closed loop control, are worth exploring. Open loop control algorithms are often the first step towards realization of gait and can already yield stable walking patterns. Open loop stable control potentially offers more robust behaviour and naturally requires little to no feedback effort compared to pure closed loop control strategies. Closing the loop of a stable open loop controller by adding sensory input is a prominent mean for cheaply improving stability and robustness. [36]

3.3.2. Central Pattern Generators

The discovery of central pattern generators in the nervous system provided a biologically inspired framework for controlling passive dynamic walkers. These *CPGs* are in the physiological sense, small autonomous neural networks that can generate rhythmic output. They control periodic motorical sequences like heartbeat, breathing, mastication, vocalization and limb movement for locomotion. The characteristics of these sequences can be influenced by sensoric feedback. [15]

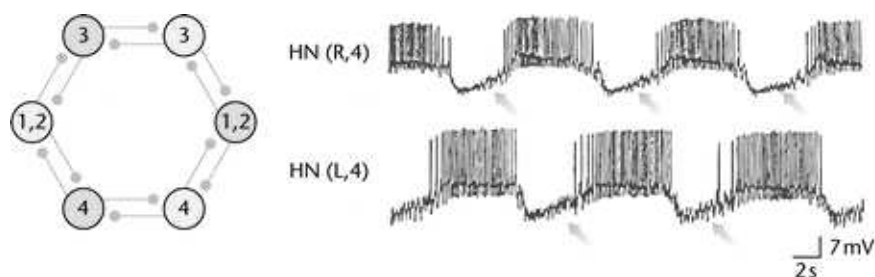


Figure 3.5.: Central pattern generator network for the leech heartbeat and its output signal. The network consists of six neurons connected by reciprocally inhibitory synapses. The neuron pairs 3 and 4 form two oscillators that burst in antiphase, coordinated through neurons 1 and 2. (from [15])

The basic principle of CPGs was adopted for humanoid robotics: a periodic signal controls the movements of the robot's limbs. The parameters of the signal generator though

can be influenced based on input from tactile, positional or acceleration sensors affecting amongst others step size and height and the interplay between legs, torso and arms. The output of the central pattern generator often is merely transformed in a simple fashion before being fed to the actuators, while enabling the desired trajectories and maintaining compatibility with the actuators. Machine Learning approaches are then a common solution for finding the “simple” transformation or “tuning” the oscillator network.

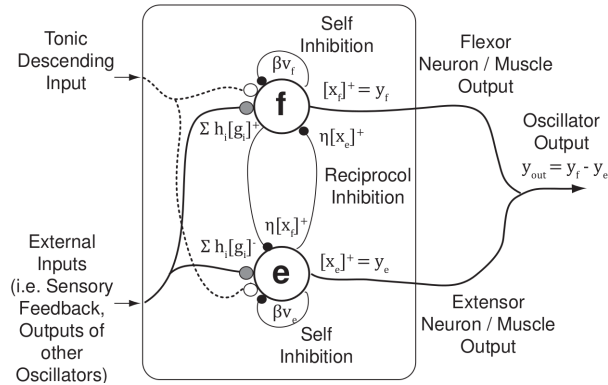


Figure 3.6.: A model of a neuronal oscillator network with two units: The Matsuoka oscillator. (figure from [16])

Generally, implementations and use of the principle of CPGs in humanoid robotics is diverse. CPGs are used not only to directly control the joints of a humanoid robot, but also to stabilize and enhance classical control strategies³ ([9], [10]). Implementations range from neural nets with neuronal units based on complex models highly inspired by neurosciences ([30]), to simple recurrent neural nets that create periodicity (see figure 3.6, [50], [9], [31]), to the interconnection of plain sinusoidal oscillators ([3], [37]) as it is also the case in this work (see chapter 6), For the latter even though there is no direct physiological template underneath anymore the term central pattern generator is still applied. With structure and oscillator type, flexibility, adaptability and also complexity of the generated control signals vary.

³i.e. CPG-joint control as opposed to CPG-trajectory control

4. Implementation: Starting Point

This chapter describes briefly, the mechanical configuration, the used electronic control and actuator units and the projects organization. The hardware was given and considered fixed at the beginning of this work. The lower limb system of the humanoid used is hereby inspired by the musculoskeletal structure of the human lower limb system. The notable difference to many other humanoids being the extensive use of cables to transfer the torque from the actuators to the joints. In fact, no lower limb joint is directly manipulated by an actuator but through the cable system.

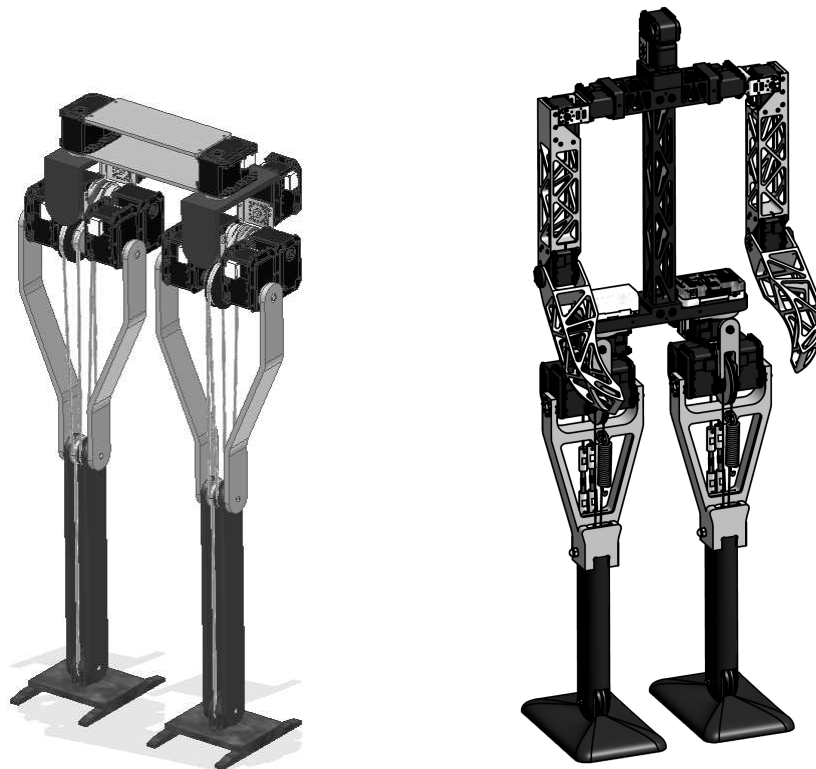


Figure 4.1.: CAD rendering of a first prototype of the lower limb system (*left*) and the robot as used in this work (*right*).

4.1. Configuration of the Robot Platform

The robot has 18 degrees of freedom: two for the head, three for each arm, two for rotation of each leg at the hip in both, coronal and transverse plane and four for the movement of each leg. Of the latter four, two are coupled and together actuate one single joint. The actuators are mainly ROBOTIS MX-28 and ROBOTIS MX-64 for the legs where higher torques are needed.

Joints and actuators for the movement of the legs are through the cables arranged in a way that torque exerted from one actuator is not directly applied on a single joint but rather influences all of them: ankle, knee and hip. This approach was named Multiple Joint Mixed Actuation (*MjMA*) in [35].

Two main aspects motivated this technique: The existence and proven importance of biarticulate muscles for bipedal locomotion (see section 3.2.2) and the simplification of control resulting from the intelligent (de-)coupling of actuators and joints.

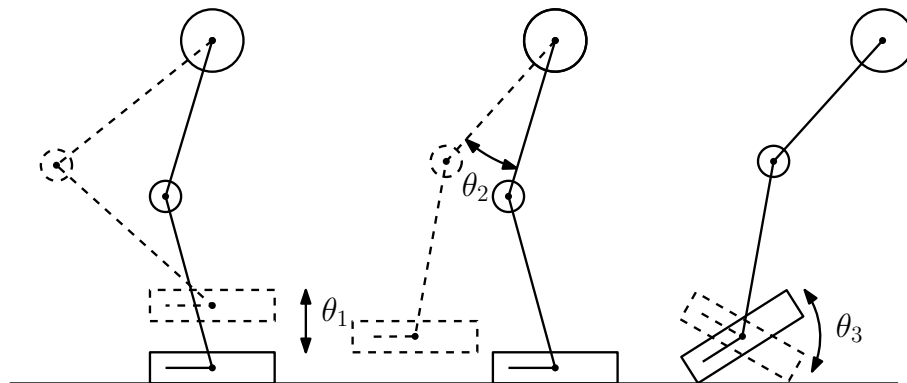


Figure 4.2.: DOF of the Lower Limb System of the Robot. The carefully devised cable transfer system allows to control relevant joint positions directly instead of having to determine the joint position by inverse kinematics.

The in [35] outlined mechanical structure obviates the solution of the inverse kinematics problem by coupling the actuators directly to configurations of the limbs essential for bipedal walking. As such,

- two actuators combined control the length of the leg by bending at knee and hip (θ_1 or *leglength*),

- one actuator controls the inclination of the whole leg (θ_2 or *kneebias*) and
- one actuator controls the inclination of the foot relative to the ground (θ_3 or *ankle*)

mostly independent of each other (see figure 4.2). This is realized by carefully designing and choosing the free and driving wheels of the cable transfer mechanism between each actuator and joint.

Further joints enable rotation of the legs around the hip in both, the coronal and the transverse plane and movement of the arms (3 each). For higher level behavior, the robot was further equipped with a HaViMo2.5 vision module at the head and a CH Robotics UM6 Orientation Sensor in the torso.

4.2. Description of the Controller Unit and Actuators

The electronic control unit at hand is the ROBOTIS CM-530. This device is equipped with a STMicroelectronics STM32F103RE Cortex-M3 Micro Controller Unit with 512 Kbytes Flash memory, 64 Kbytes SRAM, and CPU with 72 MHz maximum clock rate. The CM-530 has signal LEDs and buttons for rudimentary communication with the user, USART serial ports for the connection of additional peripherals and several ports for DYNAMIXEL actuator network links. A mini USB port with a serial to USB converter allows serial communication for installation of firmware and debugging.

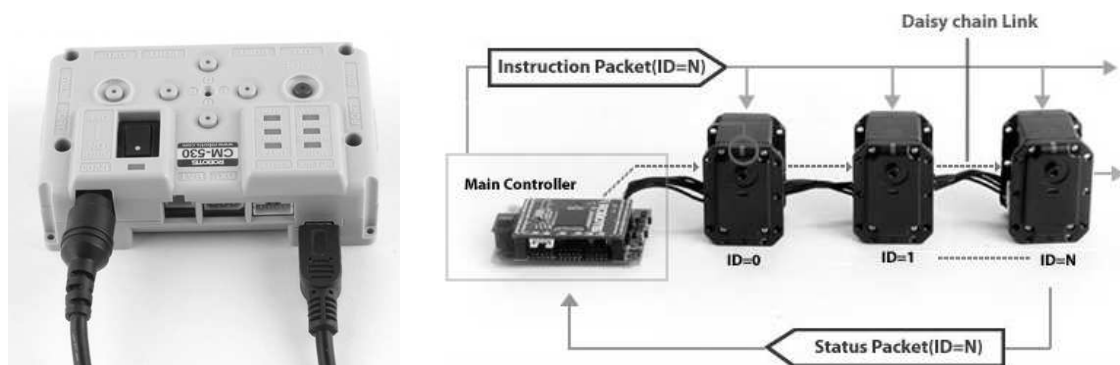


Figure 4.3.: The CM530 controller unit and structure of the DYNAMIXEL actuator network (images from ROBOTIS website <http://support.robotis.com/en/>).

The DYNAMIXEL (DXL) actuator network system consists of a series of actuators that understand the DXL communication protocol. DXL allows to connect multiple actuators

to a main controller in a daisy chain fashion: All communication participants are on a single bus and have an unique designator to be addressed by the controller. The topology of the network is hereby free to chose. The DXL control protocol is based on half-duplex USART. For using the protocol, there exists a SDK with a C API whose relevant methods are listed in table 4.1.

| Method | Parameters | Function |
|---|---|--|
| <code>dxl_ping</code> | <i>ID</i> | Check existence of actuator <i>ID</i> |
| <code>dxl_read_byte</code> <code>dxl_write_byte</code> | <i>ID, address</i> <i>ID, address, value</i> | Get/Set 8 bit value of register <i>address</i> for actuator <i>ID</i> |
| <code>dxl_read_word</code> <code>dxl_write_word</code> | <i>ID, address</i> <i>ID, address, value</i> | Get/Set 16 bit value of register <i>address</i> for actuator <i>ID</i> |
| <code>dxl_get_result</code> | - | Get result of last transmission |

Table 4.1.: Higher level communication methods of the DYNAMIXEL C API. Where *ID* is the designator of the actuator addressed and *address* the register that is been written to or read from. These registers hold all relevant information like target and current speed or torque, voltage supply, etc.

4.3. Tools and Workflow in the Project

For the CM530 in conjunction with the DXL actuator system the manufacturer provided a set of example programs, that each implement a certain aspect of the capabilities of the CM530. At the beginning of this work, the controller code was largely based on that example code.

Apart from the walking algorithm, development would have had to take place for three other necessary subsystems of autonomous robotics:

- Vision,
- Communication and
- Behavior.

Tasks that do not directly affect the implementation of the gait control but still are supposed to take place on the controller unit and even access the DXL bus. An overview of

the architecture and overall control flow as finally used in this work is given in section 5.3.

The code is organized as a Makefile project thus all eventual collaborators had free choice of the used IDE for development. This is relevant, as the MATLAB Code generator allows for complete integration of the build process. As convenient as that would be, it would make MATLAB necessary for all collaborators to build the firmware binary. Thus choice has been made, to keep the Makefile project structure of the project and use MATLAB/Simulink solely as modeling and code generation auxiliary tool for the walking algorithm. The Makefile was altered to automatically account for every artifact the code generation process produces.

The general development has taken place in an iterative *conceive, code and evaluate/debug* sort of way. Larger scale methodology as covered in section 2.2 does not entirely apply, yet some aspects of the model based paradigm certainly benefit the workflow as sketched in figure 4.4.

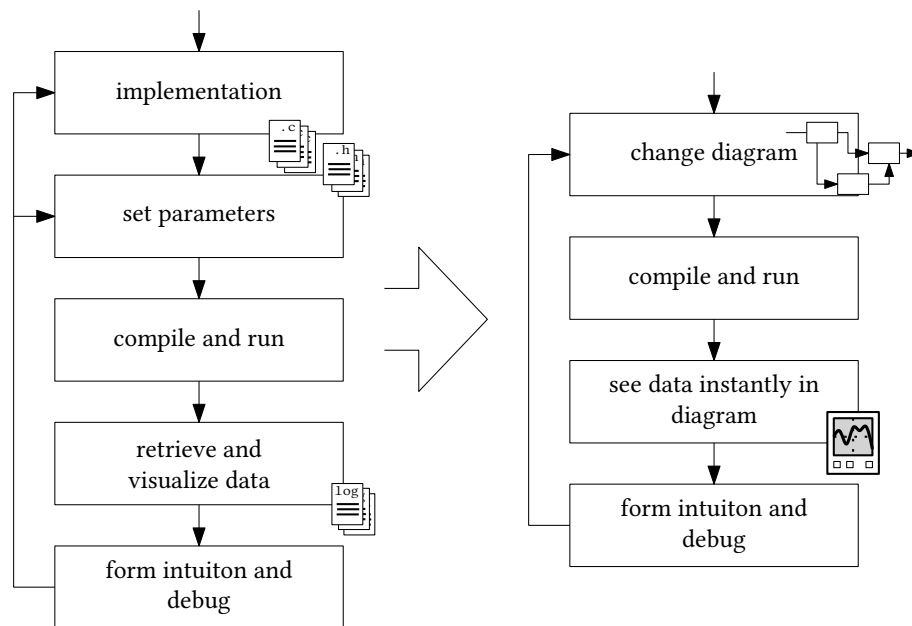


Figure 4.4.: Sketch of the walking algorithm development workflow. Two main differences are introduced by the implementation of graphical modeling: Change of the algorithm not by manual coding in C but by changing a considerably clearer control flow diagram and the ability to directly view data from the sensors and actuators in the same user interface the algorithm's diagram is shown.

5. Implementation: A Simulink Blockset for Dynamixel Peripherals

Two main aspects have to be considered for the implementation of the SIMULINK blockset: The new interfaces introduced to the graphical modeling environment, i.e. which new blocks to create and how the algorithms modeled are integrated into the controller software i.e. the interface for the firmware backend.

The code generated by Simulink Coder consists of three central routines:

- the model *initialization* routine `model_initialize()`,
- the model *step* routine `model_step()`,
- a template *main*-loop that calls the aforementioned routines.

This implies the interface the surrounding firmware program has to the model's generated code: *Initialization* to be called once at startup and the *model-step* to be called at –by the model– well defined intervals. The generated template main-loop hereby outlines the procedure. How the timing affordances are ensured is finally left to the implementation of the main loop. How this is done for the FUB-KIT project is detailed in the last section of this chapter.

The new blockset mainly maps the relevant methods of the DXL C API (see table 4.1). Built upon these base blocks were some “convenience” blocks implemented directly in Simulink that simplify the interaction with the actuators. The blocks are described in the following sections.

5.1. Description of the Implemented Blocks

For the blockset to work the existence of the methods in table 4.1 is essential. References to these methods are encapsulated in a thin abstraction layer in form of a header file (`dxl_blockhal.h`). Table 5.1 shows a list of the blocks with a short function description and figure 5.1 a view of the blockset as it appears in Simulink.

| Block Name | Function |
|-------------------------|---|
| read From DXL | Map to <code>dxl_read_byte</code> and <code>dxl_read_word</code> |
| write to DXL | Map to <code>dxl_write_byte</code> and <code>dxl_write_word</code> |
| run on Target | Custom C code statement |
| <i>Derived blocks:</i> | |
| Pseudosigned | Convert non standard signed DXL values to natively signed |
| Triggered Input Bias | Auxiliary block to conveniently bias a signal |
| Biased Motor | Deviate actuator around bias using <i>Triggered Input Bias</i> block instead of setting position directly |
| Biased Motor Springlike | Same as <i>Biased Motor</i> block but with spring-like behavior |

Table 5.1.: List of Blocks in the DYNAMIXEL Blockset

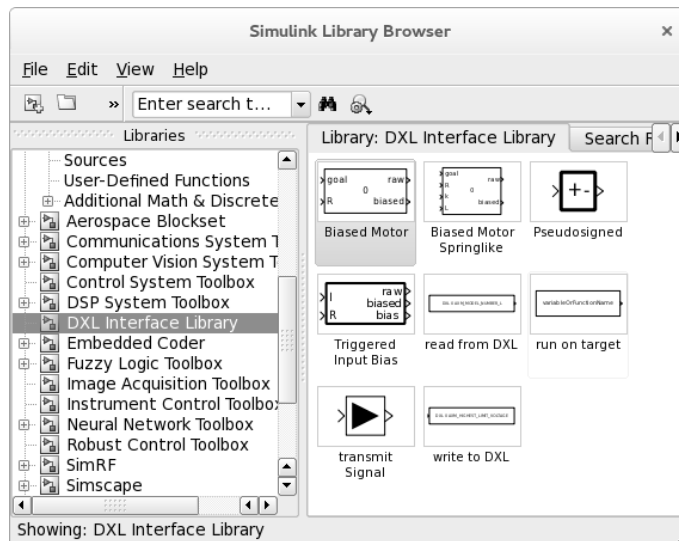


Figure 5.1.: View of the DYNAMIXEL Blockset in the Simulink Library. The transmit Signal block is described in appendix A.

5.1.1. Core Blocks

Read From and Write To DXL

Two blocks form the central elements of the blockset: *write to DXL* and *read from DXL*. The parameter windows of both blocks are essentially identical (see figure 5.2): *ID*, *Address* and *Datatype*. The *write* block accordingly has an input port but no output and vice versa for the *read* block. The list of available addresses is determined by parsing an easily extensible headerfile (`dynamixel_address_tables.h`) that is also used by the firmware backend.

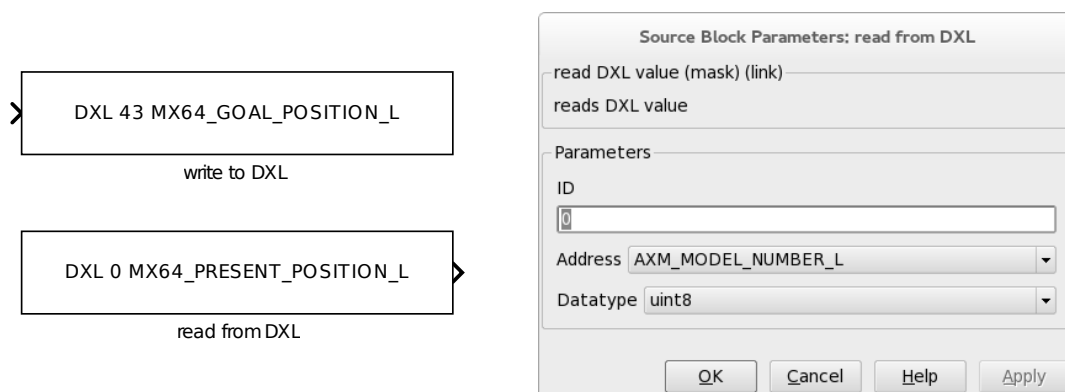


Figure 5.2.: Read and Write to Dynamixel Blocks and Parameters Window

The C code belonging to these blocks implements two buffers that hold written and read values of the model as well as a routine that dispatches the calls to the DYNAMIXEL C API (`void dxl_interface_update()`). This setup was chosen to separate the model step computation from the communication with the actuators. On the Simulink side, the Block is implemented as a MATLAB S-Function handling input ports and data types.

Run on Target

This block was implemented, to easily access functions and variables from the custom C code of the underlying firmware program. It essentially allows to define a single line function call or variable assignment offering flexibility by verbatim embedding of the user provided expression in the code. This is a convenient way to access sensor data using function calls that are already implemented in the underlying firmware.

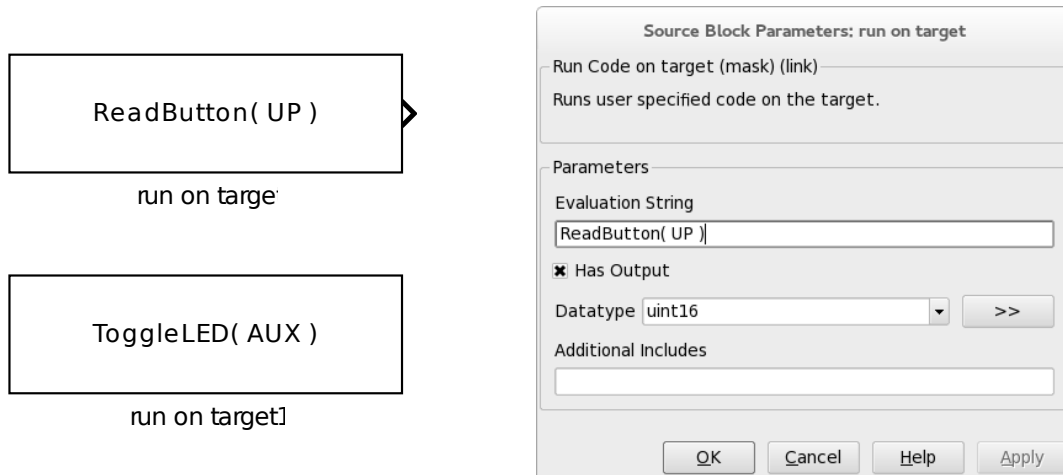


Figure 5.3.: Two Run on Target Block examples and Parameters Window

5.1.2. Derived Blocks (Subsystems)

Building upon the read from and write to DXL blocks, 3 additional blocks were implemented, directly in Simulink, to simplify programming of the actuators.

Pseudosigned

DXL actuators offer a range of state variables with varying resolution and value range (e.g. present speed of 0-1023 represents counter clock wise rotation, 1024-2047 clockwise rotation speed, similar for present position and torque values). However the returned value is always of unsigned integer data type. This block was implemented to conveniently handle these kind of values.

Triggered Input Bias

In humanoid robotics joints often simply deviate from a neutral, stable joint configuration (e.g. upright standing position). This Block allows for setting the pivot point of an actuator by triggering with a second signal. It has no parameters to be set.

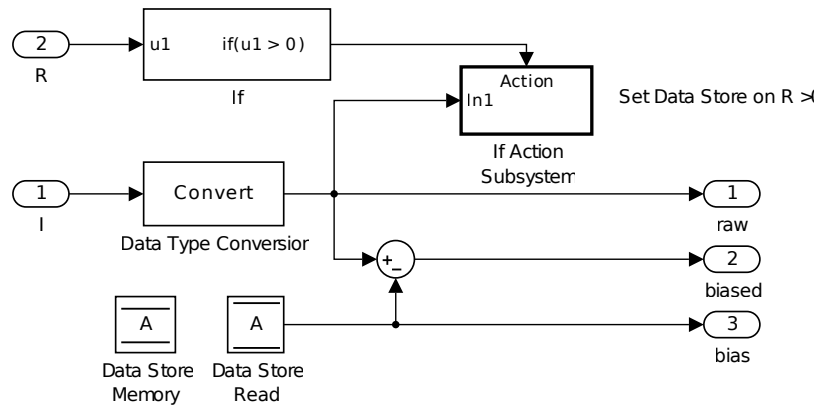


Figure 5.4.: The Triggered Input Bias Block. On the left side there are the two input ports for the signal (I) and the reset trigger (R). Output are the original signal, the bias value and the biased signal. Upon receiving a positive signal on R the current input I is written to a “Data Store Memory”. In generated code this translates to a state variable of the specified type.

Biased Motor

Built around the Triggered Input Bias block, this block directly represents a DXL actuator deviating from a set pivot point. This abstraction is the main way DXL actuators were accessed in the next chapter. The only parameter to this Block is the DXL ID of the addressed actuator.

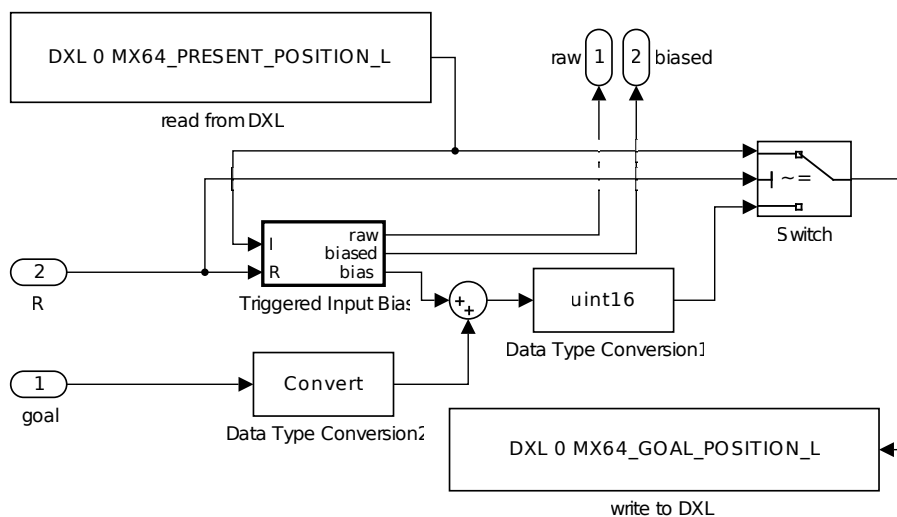


Figure 5.5.: The Biased Motor Block built around the Triggered Input Block. Input ports are again the reset trigger for setting the pivot position as well as the goal value by which the actuator is to deviate about the pivot point.

5.2. Code Generation

Specifying code generation rules is necessary for the three newly implemented core blocks. The DXL interface blocks share a common Simulink implementation and target language compiler file. Figure 5.6 exemplarily follows some parameters from the block to the generated C code in snippets of the blocks' implementation. Structure and flow is the same for all implemented blocks as shown in section 5.1.1.

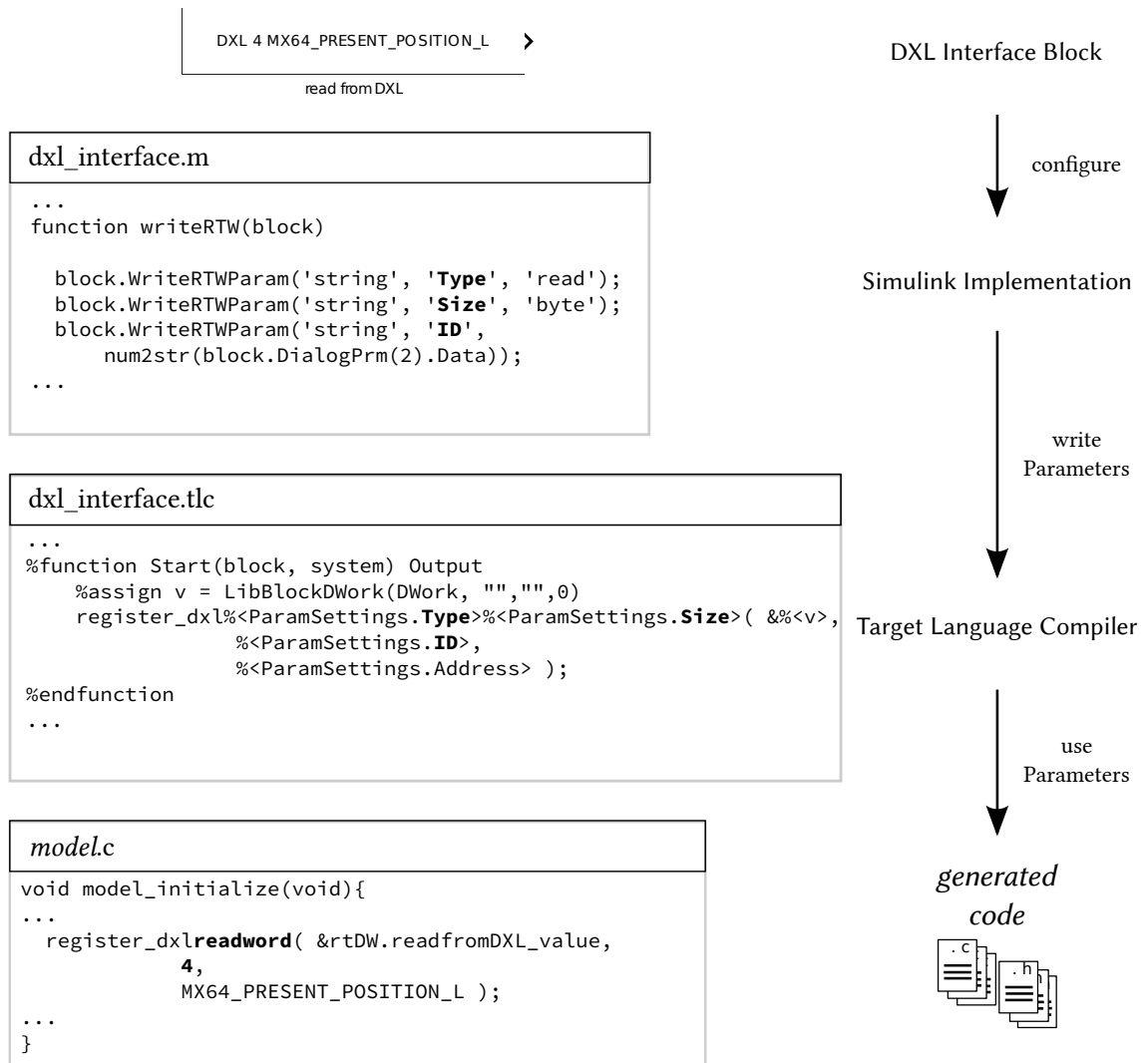


Figure 5.6.: Code generation procedure in excerpts for the *read from DXL* block. Visible, an excerpt of the additional *writeRTW* method as mentioned in 2.4.2, of the `.tlc`-file for the block and of the generated code's initialization routine for the model.

5.3. Firmware Architecture

Based upon the sample code of ROBOTIS, the processors manufacturers Standard Peripherals Library STM3210x in version 2.0.3. was used as well as a modification of Matthew Paulishen’s CM530 C/C++ “easy functions” library [40] to easily access the CM530’s functionality. This library includes the DXL C API as well as convenient routines to access amongst others LEDs, buttons, timers and serial interface. Figure A.1 shows the overall architecture of the firmware program. The marked parts in the upper right corner are the code contributions by the blockset. Only the *Walking* part is realized with Simulink model generated code. The remaining parts as well as the mainloop is in custom C code.

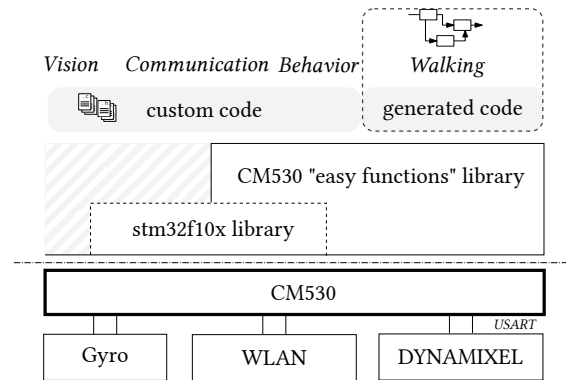


Figure 5.7.: Source code structure of the firmware program.

The structure of the firmware is a simple control loop that sequentially processes each component. The update of the walking model is done inside the interrupt service routine of a timer of the microcontroller. The timer period matches the step time of the Simulink model. Dispatching of the control values to the actuators within the ISR proved difficult due to the interrupts caused by the DXL communication routines even though the microcontroller allows for nested interrupts. As a “hack” the dispatching via `dxl_update_interface()` was pulled back into the main loop. Figure 5.8 shows the activity diagram for the main loop.

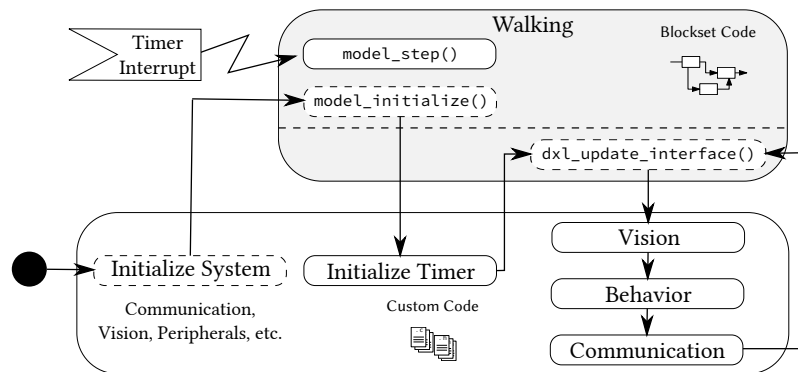


Figure 5.8.: The firmware’s main loop activity diagram, divided into custom and generated blockset code.

6. Implementation: Simulink Models to Control the Robot

To employ the toolbox and new development process, two basic walking algorithms have been implemented. Both rely on the CPG principle while using sinusoidal oscillators as pattern generator. The pattern generator is implemented as subsystem in the walking algorithm model generally remaining interchangeable between the two models. The actuators were addressed via the Biased Motor subsystem as described in the previous chapter. The controllers ran on the CM530 at a model step size of $0.02s$ ($50Hz$). Figure 6.1 shows the basic architecture of the CPG-based control algorithms.

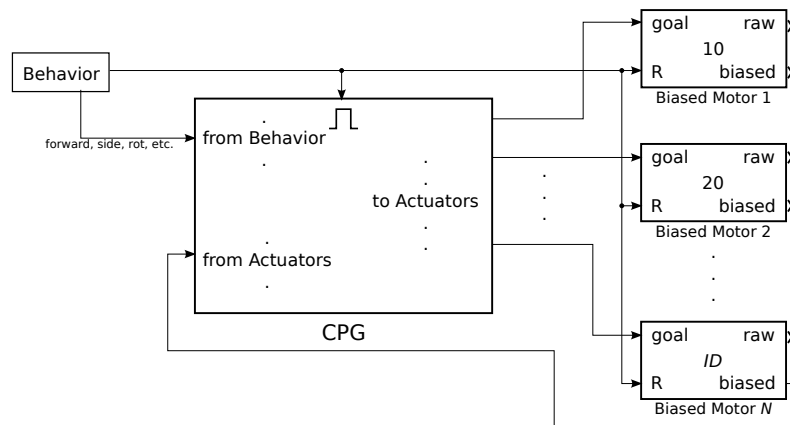


Figure 6.1.: Model diagram of the general CPG walker architecture. The *Behavior* subsystem inputs control signals to the CPG which in turns gives output to the actuators. These return, in a closed loop controller, feedback to the CPG.

Unfortunately, no stable gait pattern was achieved in the course of this work. The implemented control algorithms merely serve as a proof of concept for the DXL blockset in addition with the surrounding firmware. They only can be considered a modeling example or a base for the development of a working control algorithm.

6.1. Open Loop CPG Control

The first implemented control algorithm is based on the open loop control algorithm as proposed in [35]. The control signals derived for the CPG there were as follows:

$$\begin{pmatrix} \text{leglength}_l \\ \text{leglength}_r \\ \text{kneebias}_l \\ \text{kneebias}_r \end{pmatrix} = \begin{pmatrix} \max(r \sin(\omega t), 0) \\ \max(-r \sin(\omega t), 0) \\ v_x \sin(\omega t + \phi_x) \\ -v_x \sin(\omega t + \phi_x) \end{pmatrix}$$

This addresses joints θ_1 (*leglength*) and θ_2 (*kneebias*) of figure 4.2. The translation of these equations to a graphical model in Simulink is shown in figure 6.2. It merely implements a simple stepping motion of the lower limbs and accounts for a specified forward walking speed v_t . Figure 6.3 shows the signals produced by this controller as well as –for qualitative comparison– by an evaluation of the model in Simulink.

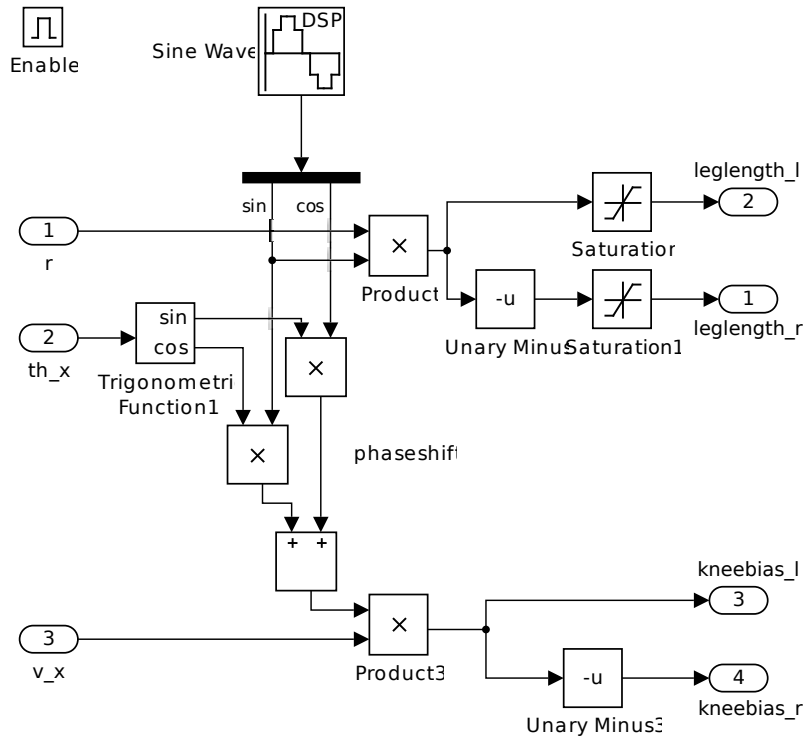


Figure 6.2.: The graphical implementation of the above expressions into the CPG subsystem embedded as depicted in figure 6.1. The signals v_x , th_x and r would come from the behaviour subsystem.

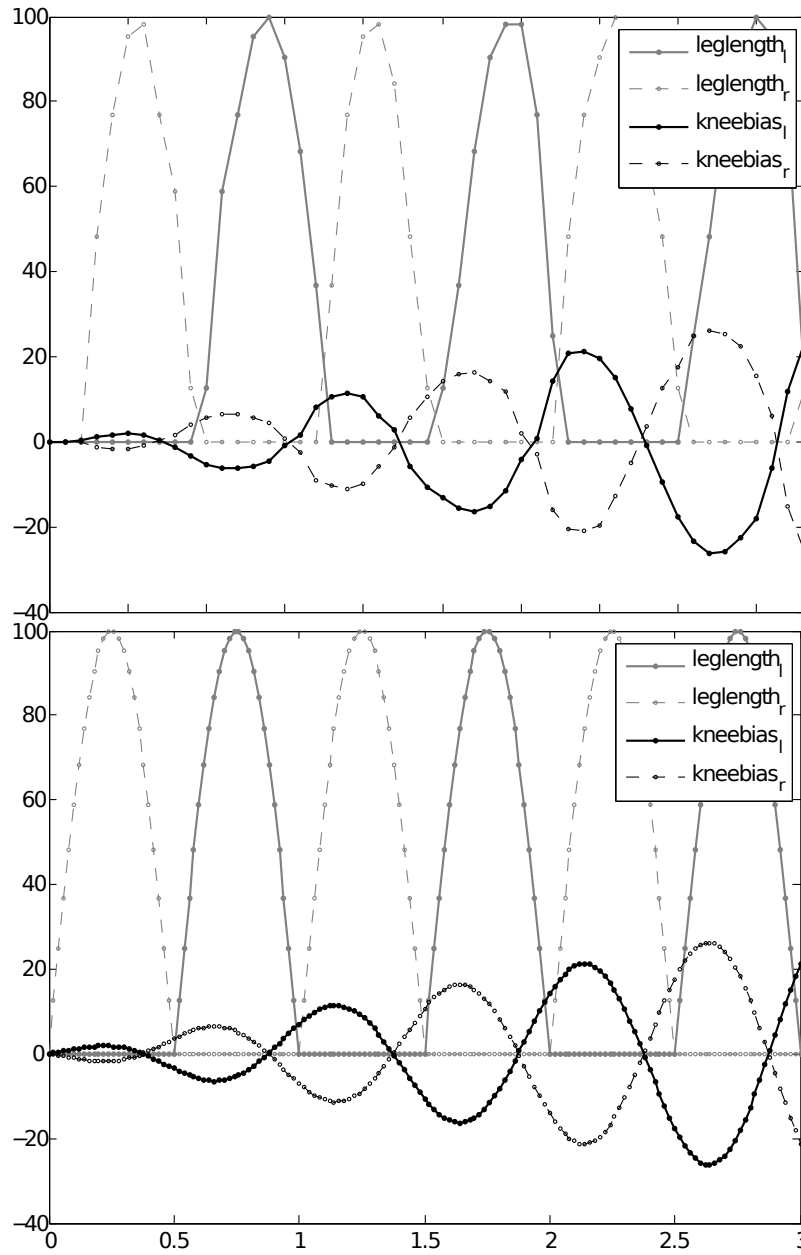


Figure 6.3.: The control signals for the open loop controller using $\theta_x = \pi/4$, $r = 100$ and $\omega = 2\pi$. Forward speed v_x was increasing with slope $10/s$. *Below* as evaluated purely in Simulink, *above* as produced by the controller and captured using the mechanism described in appendix A. Only a qualitative validation would be possible, as said mechanism provides no useful timescale.

6.2. A “Weakly” Closed Loop CPG

The second CPG model was derived from the source code of the walking algorithm of the 2013 RoboCup contribution of the FUB-KIT project. Basically, the C code was directly translated to the Simulink model which is shown in figure 6.4.

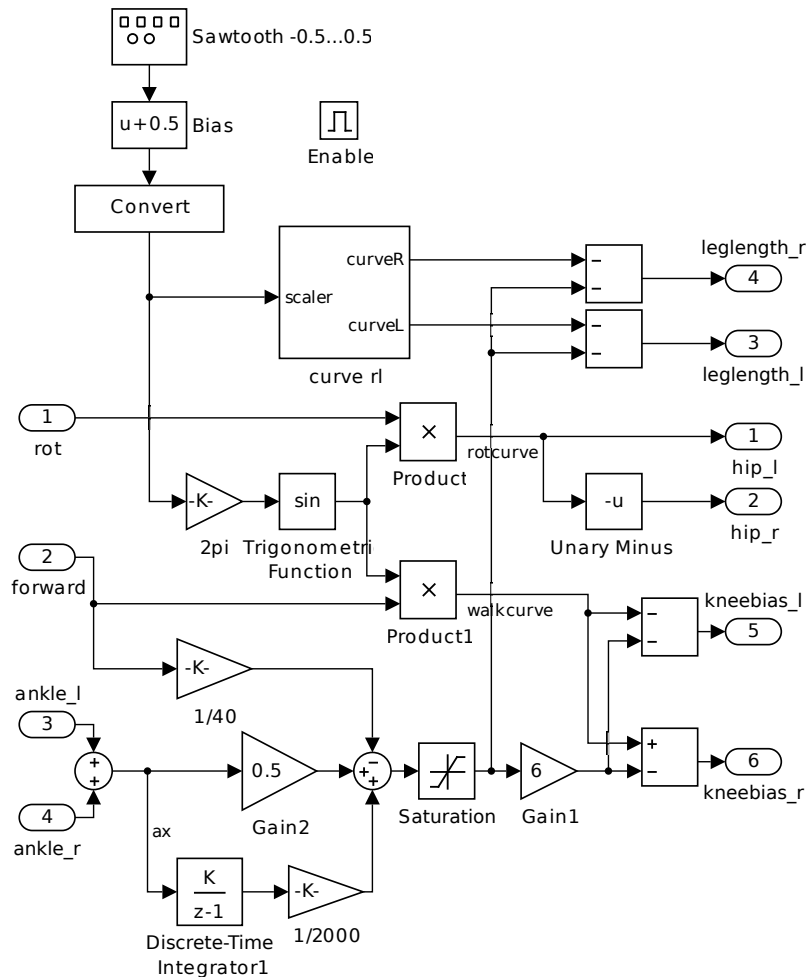


Figure 6.4.: Graphical implementation of the walking algorithm for the FUB-KIT project from RoboCup 2013. Feedback from the ankle actuators was intended to stabilize the walking signals (hence “weakly closed”).

The subsystem *curve rl* realizes the following mathematical expressions:

$$curveR = \begin{cases} 200 \cdot \sin \pi \sqrt{2 \cdot scaler - 1} - 50 & \text{if } scaler > 0.5 \\ -50 & \text{else} \end{cases}$$

and

$$curveL = \begin{cases} 200 \cdot \sin \pi \sqrt{2 \cdot scaler} - 50 & \text{if } scaler < 0.5 \\ -50 & \text{else} \end{cases}$$

which essentially derives two alternating sinusoidal signal courses from the sawtooth signal *scaler*. The graphical implementation results in a spacious diagram and shall be omitted here. As in the previous section, figure 6.5 shows produced control signals.

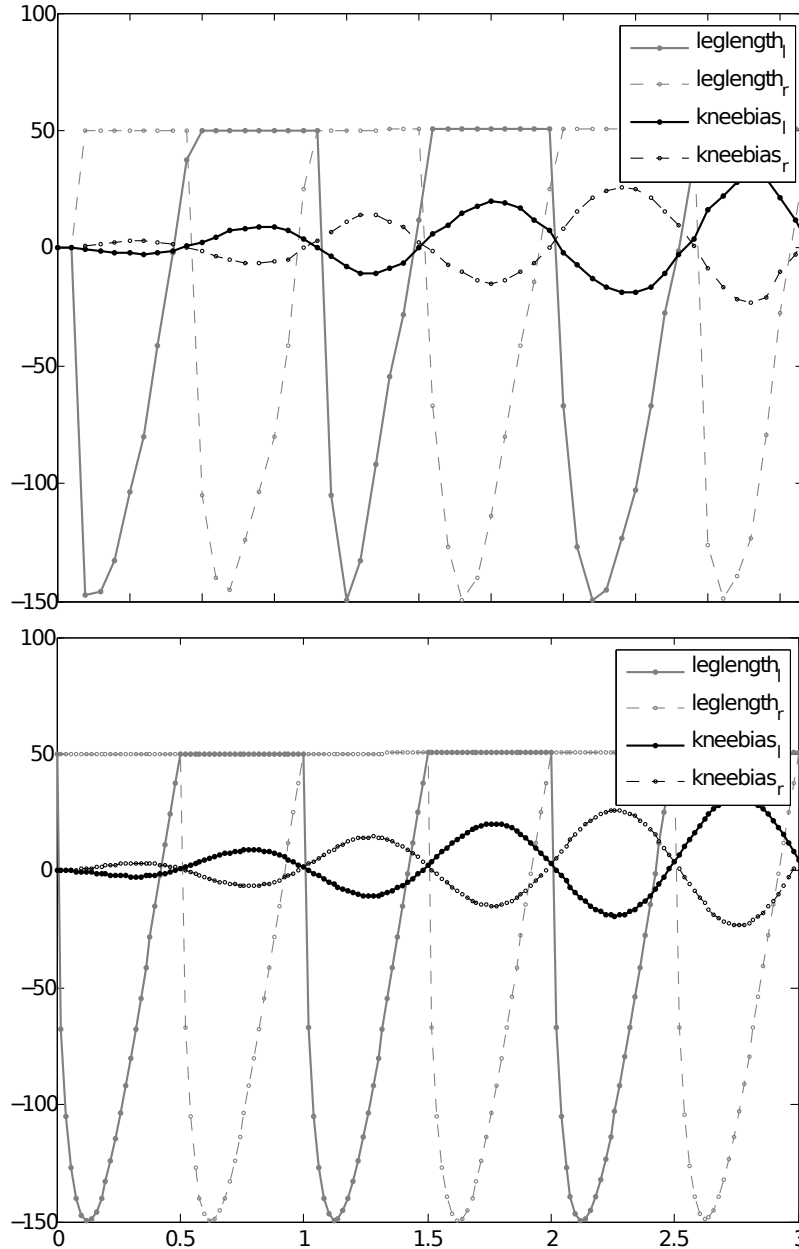


Figure 6.5.: The control signals for the controller based on the RoboCup 2013 code. Forward speed (“forward”) was increasing with slope 10/s. No rotation was set and the feedback signal was set to 0. Analogously to figure 6.3, *below* as evaluated purely in Simulink and *above* as produced by the controller.

7. Conclusion & Outlook

The goal of this work was to enable graphical modeling for programming of the FUB-KIT robot as well as to provide an example algorithm in the respective graphical modeling environment. This goal has been achieved in a rudimentary fashion: Firmware for the control unit has been developed to support the code generated by the graphical modeling tool; The graphical modeling tool has been customized as to conveniently produce supported code for the underlying firmware; and in a minimalistic way two sample control algorithms have been implemented graphically up to the point where the functioning of the three parts *firmware*, *modeling tool* and *control algorithm* was shown.

The graphical implementation of control algorithms turned out ambivalently. While the MATLAB/Simulink combination offers a comfortable IDE for the design of such algorithms, a considerable amount of familiarization with the product is necessary to fully take advantage of its capabilities. This especially applies to code generation for uncommon platforms and when inherent limitations and peculiarities of the target have to be considered. This may be not that much the case when working with officially supported platforms and ideally is a one-time effort as has been done in this work. As soon as this ground work is done the use of a modeling tool like Simulink can doubtlessly enhance development efficiency for the design of walking control algorithms.

A minor cosmetic drawback general to dataflow programming is the at times as inconvenient resulting graphical implementation of otherwise simple expressions. A oneliner in C eventually turns out to be a rather convoluted flow diagram. The abstraction into subsystems naturally compensates this detail. Further benefits like the comfortable access to sensor data in the IDE and the possibility to quickly prototype, implement and evaluate control signal patterns certainly outweigh the drawbacks. The simplistic execution of this work however bears a lot of space for improvement as well as possibilities for extension. Some angles are mentioned in the next paragraphs.

Optimize Walking Algorithms The mentioned benefits of using a graphical modeling tool for the development of control algorithms have hardly been harnessed during this work: The improved convenience in design, parameter tuning and optimization. Models shown in chapter 6 were implemented on the spot and configured with sane default parameters that enabled basic functioning. Little to no effort was made to achieve a stable walking pattern. Easing the way to get there, was the primary motivation for the implementation of graphical modeling. Enabling easy and direct access to sensor data should further facilitate debugging and development in the course. Eventually, MATLAB/Simulink offers a comfortable environment to quickly devise more sophisticated control algorithms.

Integrate Simulation Simulink is in the first place a simulation environment for dynamic systems. Only a glimpse at the potential it offers for the design of control algorithms was taken, when it was used to merely evaluate the control signal course of the implemented algorithms in chapter 6. Two approaches to involve simulation come to mind:

- Any simulation can be realized directly in Simulink, however, completely modeling a humanoid robot (not only a forward control algorithm) in Simulink certainly is a tedious task,
- The structure of the toolbox suggests to implement an interface to an external robotic simulation environment like *gazebo*. As mentioned, the blocks implemented in this work only are functional during the code generation process. Equipping these blocks with said interface to allow for simulation in Simulink would be useful.

Enabling Simulation would complete the Model Based Design principle: Control algorithms could be modeled and validated in simulation, before being dispatched to the controller hardware. An intermediate step possibly worth exploring could be the use of USB2Dynamixel with the host computer to test and debug the control algorithms while running on the host.

Integrate Behavioral Control Focus of this work lies in the modeling of continuous dynamics and controlling the movement of the limbs of the humanoid robots. Another aspect relevant to autonomous robotics and favourably modeled or programmed graphically, is “intelligent” behavior. Behaviour is prominently modeled using state machines. The MathWorks offers a modeling utility called *StateFlow* that integrates into the MATLAB/Simulink environment. Code generation is equally supported. One possibility would be to integrate the behavior control directly inside of the already implemented models. For simple behavior this seems appropriate. For involving more sensory input and possibly communication with other agents, it would make sense to move less time critical components such as those not contributing to the dynamics of the robot to a separate controller platform.

A. Serial Communication with the Microcontroller for Debugging

MATLAB/Simulink offers a very convenient way to debug modeled control algorithms on hardware called *External Mode*. Simulink communicates with the generated firmware code over the serial bus while enabling direct view of signals in the model and the possibility of on-line changing of parameters of blocks in the model. However, the processor in the CM530 turned out to have too little memory for the *External Mode* to work properly.

To still enable debugging by viewing the signals directly in the model a *transmit signal* block was developed and added to the blockset. On the Simulink side this block accesses the serial port and requests with every model update the current value of the respective signal in memory. To correctly assign Simulink-side block to firmware-side value the block has an numerical identifier that has to be uniquely defined in the model. Due to the simplicity of the implementation the timescale is incorrect. Live viewing of the signal is possible and a convenient tool to have. However, this provisional workaround is only a marginal substitute for Simulink's *External Mode*.

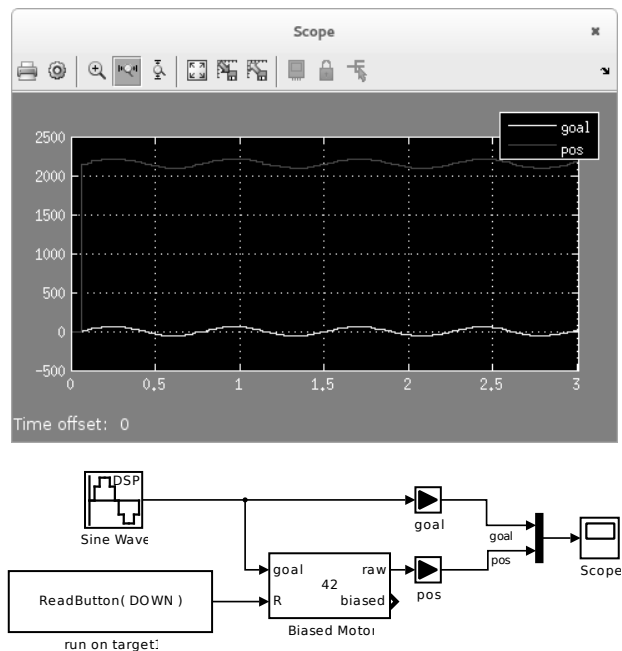


Figure A.1.: An example model employing the *transmit signal* block and a screenshot of the scope window as seen in the IDE.

B. Fixed Point Numbers and their Application in the Toolbox

Floating point operations are computationally expensive. The floating point representation with exponent and fraction entails expensive conversion procedures for simple arithmetic operations. Microprocessors or CPUs usually are built with a separate floating point unit (FPU) in order to accelerate these operations. However many embedded processors do not have this kind of hardware acceleration available as it is the case for the processor used in this work.

To still accelerate operations on real numbers without the help of a FPU they can be represented in fixed point: The binary representation consists of an integer part and a fractional part with the radix point of the number being fixed (hence fixed point arithmetics). Arithmetics with this representation are reduced to simple integer operations as opposed to the expensive operations using floating point numbers.

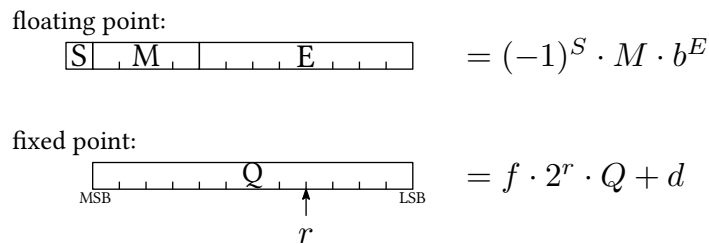


Figure B.1.: Fixed point and Floating point binary patterns. A floating point value is defined by its sign S , significand M and exponent E . The base b depends on the implementation (usually 2 or 10). The real value stored in a fixed point datatype depends on more implementation specific values, namely radix point r , scaling factor f and bias d . Signedness is treated as with integer datatypes e.g. using two's-complement (Simulink Coder requires two's-complement representation for signed integer values). The higher implementation side maintenance effort as well as a reduced value range "buy" the accelerated computation.

MATLAB/Simulink provides fixed point arithmetics functionality via the additional Fixed Point Designer toolbox which also enables support for code generation using fixed point data types. To employ this functionality in the DXL-toolbox, the datatype of all implemented blocks are by default set up via a one configurable function `dxl_fixdt()`. The default fixed point datatype is set to a signed number with scaling factor $f = 1$, no bias, word length of 32 bit for Q and the radix point at $r = 16$. This results in a value range from -32767 to 32768 with precision of $\approx 0.015 \cdot 10^{-3}$ which is more than suitable for addressing the actuators used.

C. Toolbox Setup and Source Structure

Dependencies

For building the firmware and creating control algorithms with the toolbox following Software is needed:

- Cross compiler for ARM EABI (used version 4.9.1)
Download at <https://launchpad.net/gcc-arm-embedded>
Linux distributions may offer packages.
- *make*
- MATLAB and Simulink (used version 2013b) with toolboxes
 - Simulink Coder
 - Embedded Coder
 - (DSP System Toolbox)
 - (Fixed-Point Designer)

The DSP System Toolbox provides advanced and optimized signal generators, filters, transformations and support for fast trigonometric functions. The advantages of the Fixed-Point Designer toolbox are explained in appendix B.

Directory Structure and Build Configuration

The directory structure of the project's code is as follows:

```
/
+ apps/ – general firmware source directory
    + fubkit/ – custom firmware sourcecode for this work
      Other Subsystem's C Code
      Makefile
    ...
+ cm530/ – cm530 “easy functions” and stm32f10x libraries
+ simulink/ – Simulink related source files
    + dxl-blockset/ – Simulink Blockset
    + models/ – Simulink model directory
      Firmware Backend C Code
+ tools/ – auxiliary programs
```

To add the toolbox to Simulink add `simulink/dxl-blockset` to the MATLAB path. The makefile is set up to compile for the model `fubkit.slx` in `simulink/models`; This directory also has to be MATLAB's working directory while generating code. The code generator's configuration is saved in the provided Simulink models.

Bibliography

- [1] Mohd Azizi Abdul Rahman and Makoto Mizukaw. Model-based development and simulation for robotic systems with SysML, simulink and simscape profiles. *International Journal of Advanced Robotic Systems*, page 1, 2013. ISSN 1729-8806. doi: 10.5772/55533. 11
- [2] P. Albertos. *Feedback and control for everyone*. Springer, Berlin ; London, 2010. ISBN 9783642034459. 7, 8, 12
- [3] Shinya Aoi, Yoshimasa Egi, Ryuichi Sugimoto, Tsuyoshi Yamashita, Soichiro Fujiki, and Kazuo Tsuchiya. Functional roles of phase resetting in the gait transition of a biped robot from quadrupedal to bipedal locomotion. *IEEE Transactions on Robotics*, 28(6):1244–1259, 2012. ISSN 1552-3098. doi: 10.1109/TRO.2012.2205489. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6243219. 30
- [4] Lars Bluma. Das blockdiagramm und die “systemingenieure”. eine visualisierungspraxis zwischen wissenschaft und Öffentlichkeit in der US-amerikanischen nachkriegszeit. *NTM International Journal of History & Ethics of Natural Sciences, Technology & Medicine*, 10(4):247–260, 2002. URL <http://link.springer.com/article/10.1007/BF02908785>. 9
- [5] Shi-Kuo Chang and Stefano Levialdi. Foreword. *Journal of Visual Languages & Computing*, 1(1):1–2, March 1990. ISSN 1045926X. doi: 10.1016/S1045-926X(05)80031-X. URL <http://linkinghub.elsevier.com/retrieve/pii/S1045926X0580031X>. 2
- [6] Steven H. Collins, Martijn Wisse, and Andy Ruina. A three-dimensional passive-dynamic walking robot with two legs and knees. *The International Journal of Robotics Research*, 20(7):607–615, 2001. URL <http://ijr.sagepub.com/content/20/7/607.short>. 28

- [7] Mirko Conrad, Ines Fey, and Sadegh Sadeghipour. Systematic model-based testing of embedded automotive software. *Electronic Notes in Theoretical Computer Science*, 111:13–26, January 2005. ISSN 15710661. doi: 10.1016/j.entcs.2004.12.005. URL <http://linkinghub.elsevier.com/retrieve/pii/S1571066104052302>. 13
- [8] Microsoft Corporation. VPL introduction, November 2014. URL <http://msdn.microsoft.com/en-us/library/bb483088.aspx>. 3
- [9] G. Endo, J. Morimoto, T. Matsubara, J. Nakanishi, and G. Cheng. Learning CPG-based biped locomotion with a policy gradient method: Application to a humanoid robot. *The International Journal of Robotics Research*, 27(2):213–228, February 2008. ISSN 0278-3649. doi: 10.1177/0278364907084980. URL <http://ijr.sagepub.com/cgi/doi/10.1177/0278364907084980>. 30
- [10] Soichiro Fujiki, Shinya Aoi, Tsuyoshi Yamashita, Tetsuro Funato, Nozomi Tomita, Kei Senda, and Kazuo Tsuchiya. Adaptive splitbelt treadmill walking of a biped robot using nonlinear oscillators with phase resetting. *Autonomous Robots*, pages 1–12, 2013. URL <http://link.springer.com/article/10.1007/s10514-013-9331-6>. 30
- [11] Mariano Garcia, Anindya Chatterjee, Andy Ruina, and Michael Coleman. The simplest walking model: stability, complexity, and scaling. *Journal of biomechanical engineering*, 120(2):281–288, 1998. URL <http://biomechanical.asmedigitalcollection.asme.org/article.aspx?articleid=1401236>. 27, 28
- [12] Robert D. Gregg, Timothy Bretl, and Mark W. Spong. Asymptotically stable gait primitives for planning dynamic bipedal locomotion in three dimensions. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 1695–1702. IEEE, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5509585. 27
- [13] LEGO® Group. LEGO.com – MINDSTORMS EV3, November 2014. URL <http://www.lego.com/de-de/mindstorms/products/ev3/31313-mindstorms-ev3/>. 3
- [14] Simon Philipp Hohberg. *Interactive Key Frame Motion Editor for Humanoid Robots*. Bachelor’s thesis, Freie Universität Berlin, Berlin, 2012. 3
- [15] Scott L. Hooper. Central pattern generators. *eLS*, 2001. URL <http://onlinelibrary>.

wiley.com/doi/10.1038/npg.els.0000032/full. 29

- [16] Helen J. Huang and Daniel P. Ferris. Computer simulations of neural mechanisms explaining upper and lower limb excitatory neural coupling. *Journal of neuroengineering and rehabilitation*, 7(1):59, 2010. URL <http://jneuroengrehab.com/content/7/1/59/abstract>. 30
- [17] Berlin United – Nao Team Humboldt. XabslEditor, November 2014. URL <http://www.naoteamhumboldt.de/en/projects/xabsleditor/>. 4
- [18] The MathWorks Inc. How code is generated from a model, September 2014. URL <http://www.mathworks.de/de/help/rtw/ug/how-code-is-generated-from-a-model.html>. 19
- [19] The MathWorks Inc. Introduction to the model.rtw file, September 2014. URL <http://www.mathworks.de/de/help/rtw/tlc/introduction-to-the-model-rtw-file.html>. 21
- [20] The MathWorks Inc. Introduction to the target language compiler, September 2014. URL <http://www.mathworks.de/de/help/rtw/tlc/introduction-to-the-model-rtw-file.html>. 21
- [21] The MathWorks Inc. MATLAB documentation center, September 2014. URL <http://www.mathworks.de/de/help/index.html>. 13
- [22] The MathWorks Inc. Modeling dynamic systems, June 2014. URL <http://www.mathworks.de/de/help/simulink/ug/modeling-dynamic-systems.html>. 17
- [23] The MathWorks Inc. Simulating dynamic systems, September 2014. URL <http://www.mathworks.de/de/help/simulink/ug/simulating-dynamic-systems.html>. 17
- [24] The MathWorks Inc. Simulink engine interaction with c s-functions, September 2014. URL <http://www.mathworks.de/de/help/simulink/sfg/how-the-simulink-engine-interacts-with-c-s-functions.html>. 16, 21
- [25] Thomas A. Johnson, Jonathan M. Jobe, Christiaan JJ Paredis, and Roger Burkhart. Modeling continuous system dynamics in SysML. In *ASME 2007 International Mechanical Engineering Congress and Exposition*, pages 197–205. American Society of

- Mechanical Engineers, 2007. URL <http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleid=1598872>. 10, 11
- [26] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kazuhito Yokoi, and Hirohisa Hirukawa. The 3d linear inverted pendulum mode: A simple modeling for a biped walking pattern generation. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 1, pages 239–246. IEEE, 2001. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=973365. 27
- [27] Ramtin Raji Kermani. *Model-based Design, Simulation and Automatic Code Generation For Embedded Systems and Robotic Applications*. Master’s thesis, Arizona State University, 2013. URL http://www.public.asu.edu/~gfaineko/pub/thesis/kermani_2013_ms_thesis.pdf. 4
- [28] Johannes Kulick. *Ein stabiler Gang für humanoide, Fußball spielende Roboter*. Master’s thesis, FU Berlin, Berlin, 2011. 25
- [29] Luciano Lavagno and Claudio Passerone. Design of embedded systems. In Richard Zurawski, editor, *Embedded Systems Handbook*, volume 6, pages 3–1–3–24. CRC Press, August 2005. ISBN 978-0-8493-2824-4, 978-1-4200-3816-3. URL <http://www.crcnetbase.com/doi/abs/10.1201/9781420038163.ch3>. 17
- [30] Cai Li, Robert Lowe, and Tom Ziemke. Humanoids learning to walk: A natural CPG-actor-critic architecture. *Frontiers in Neurorobotics*, 7, 2013. ISSN 1662-5218. doi: 10.3389/fnbot.2013.00005. URL <http://www.frontiersin.org/Neurorobotics/10.3389/fnbot.2013.00005/abstract>. 30
- [31] Guang Lei Liu, Maki K. Habib, Keigo Watanabe, and Kiyotaka Izumi. Central pattern generators based on matsuoka oscillators for the locomotion of biped robots. *Artificial Life and Robotics*, 12(1-2):264–269, April 2008. ISSN 1433-5298, 1614-7456. doi: 10.1007/s10015-007-0479-z. URL <http://link.springer.com/10.1007/s10015-007-0479-z>. 30
- [32] Holger Lutz and Wolfgang Wendt. *Taschenbuch der Regelungstechnik: mit MATLAB und Simulink*. Harri Deutsch, Frankfurt am Main, 2012. ISBN 9783817118953 3817118953 9783808556788 3808556781. 7, 9

- [33] M. Löttsch, M. Risler, and M. Jüngel. XABSL - a pragmatic approach to behavior engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129, Beijing, China, October 2006. 4
- [34] Hamid Mobalegh. *Development of an Autonomous Humanoid Robot Team*. PhD thesis, Freie Universität Berlin, Berlin, 2011. URL http://www.diss.fu-berlin.de/diss/servlets/MCRFileNodeServlet/FUDISS_derivate_000000010674/thesis_24_12_2011.pdf?hosts=local. 26, 27
- [35] Hamid Reza Mobalegh. Multi joint mixed actuation: a biologically inspired bipedal robot leg architecture. *Unpublished*. 32, 44
- [36] Katja D Mombaur. *Stability optimization of open loop controlled walking robots*. VDI-Verl., Düsseldorf, 2002. ISBN 3183922088 9783183922086. 26, 29
- [37] Jun Morimoto and Christopher G. Atkeson. Nonparametric representation of an approximated poincaré map for learning biped locomotion. *Autonomous Robots*, 27(2):131–144, September 2009. ISSN 0929-5593, 1573-7527. doi: 10.1007/s10514-009-9133-z. URL <http://link.springer.com/10.1007/s10514-009-9133-z>. 30
- [38] Oliver Alt. *Modell-basierte Systementwicklung mit SysML*. 2012. ISBN 9783446430662. 12
- [39] Florent Ouchet. Matlab simulink library for the darwinop-ens robot, November 2014. URL <https://github.com/darwinop-ens/simulink>. 4
- [40] Matthew Paulishen. CM530 c/c++ "easy functions". URL <https://github.com/tician/cm530>. 42
- [41] José Luis Pino, Soonhoi Ha, Edward A. Lee, and Joseph T. Buck. Software synthesis for DSP using ptolemy. *Journal of VLSI signal processing systems for signal, image and video technology*, 9(1-2):7–21, January 1995. ISSN 0922-5773, 1573-109X. doi: 10.1007/BF02406468. URL <http://link.springer.com/article/10.1007/BF02406468>. 19
- [42] Katayon Radkhah. *Advancing Musculoskeletal Robot Design for Dynamic and Energy-Efficient Bipedal Locomotion*. PhD thesis, TU Darmstadt, 2013. 27

- [43] Mark E. Rosheim. *Leonardo's lost robots*. Springer, Berlin, 2006. ISBN 3540284400. 1
- [44] Aldebaran Robotics SAS. SDK, simple software for developing your NAO, November 2014. URL <http://www.aldebaran.com/en/robotics-solutions/robot-software/development>. 3
- [45] Jonathan Sprinkle, J. Mikael Eklund, Humberto Gonzalez, Esten Ingar Grøtli, Ben Upcroft, Alex Makarenko, Will Uther, Michael Moser, Robert Fitch, Hugh Durrant-Whyte, and S. Shankar Sastry. Model-based design: a report from the trenches of the DARPA urban challenge. *Software & Systems Modeling*, 8(4):551–566, September 2009. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-009-0116-5. URL <http://link.springer.com/10.1007/s10270-009-0116-5>. 3, 12
- [46] Ingo Stuermer, Mirko Conrad, Heiko Doerr, and Peter Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9): 622–634, September 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70708. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4288195>. 12, 19
- [47] Andres Toom, Tonu Naks, Marc Pantel, M Gandriau, and I Wati. Gene-auto: an automatic code generator for a safe subset of SimuLink/StateFlow and scicos. In *European Conference on Embedded Real-Time Software (ERTS'08)*, 2008. 19
- [48] Tim Weilkiens. *Systems Engineering mit SysML/UML Modellierung, Analyse, Design*. dpunkt, Heidelberg, Neckar, 2014. ISBN 9783864900914 3864900913. 10
- [49] Dirk Wollherr. *Design and control aspects of humanoid walking robots*. PhD thesis, VDI-Verl., Düsseldorf, 2005. 8, 26, 27
- [50] Woosung Yang, Hyungjoo Kim, and Bum Jae. Biologically inspired self-stabilizing control for bipedal robots. *International Journal of Advanced Robotic Systems*, page 1, 2013. ISSN 1729-8806. doi: 10.5772/55463. URL http://www.intechopen.com/journals/international_journal_of_advanced_robotic_systems/biologically-inspired-self-stabilizing-control-for-bipedal-robots. 30
- [51] Volker Zerbe. Vorlesungsskript systemtheorie TU ilmenau, 2009. 6